



CECOM

CENTER FOR SOFTWARE ENGINEERING
ADVANCED SOFTWARE TECHNOLOGY

CLEARED
FOR OPEN PUBLICATION

DEC 19 1989 12

DIRECTORATE FOR FREEDOM OF INFORMATION
AND SECURITY REVIEW (OASD-PA)
DEPARTMENT OF DEFENSE

Subject: **Final Report - Catalogue of Ada Runtime
Implementation Dependencies**

CIN: C02 092JB 0001

15 FEBRUARY 1989

DTIC
ELECTRONIC
JUN 22 1990
S
E *mv*

895374

90 06 21 038

AD-A223 084

Catalogue Of Ada
Runtime Implementation
Dependencies



PREPARED FOR:
U.S. Army HQ CECOM
Center for Software Engineering
Advanced Software Technology
Fort Monmouth, NJ 07703-5000

PREPARED BY:
LabTek Corporation
8 Lunar Drive
Woodbridge, CT 06525

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

DATE:
15 October 1988

The views, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Table of Contents

1. Preface	1
2. Rationale	2
3. Reader's Guide	4
4. Performance Issues	6
Elaboration Checks (Elimination of)	6
Storage Reclamation (Time of)	7
Exceptions During Storage Allocation	8
Null Statement (Code Generated for)	9
Case Statement Implementation	10
Elaboration of Block Statements	12
Storage Allocation for Objects (Static)	13
Parameter Passing Mechanism	14
Parameter Passing Conventions	15
Pragma Inline	16
Exception Raise Overhead	17
Exception Handler Overhead	18
Error Detection, Efficiency	19
Pragma Suppress	20
Machine Code Inserts	21
Temporary File Status	21
Pragma Inline	22
Pragma Suppress	22
5. Storage Issues	23
Representation of a Literal	23

Table of Contents

Aggregates (Representation of)	24
Library Unit (Unreferenced Elements of)	25
Size Specification for Composite Types	26
Storage Reserved for a Collection Size	26
Storage Reserved for a Task Activation	27
Expressions in Enumeration Representation Clauses	27
Limitations on Record Alignments	28
Overlapping Storage Boundaries in Record Representation Clauses	28
Storage Layout for Record Types	29
Runtime Overlays	29
Storage Layout for Array Types	30
Size of a Direct Access File	31
Pragma Optimize	32
Pragma Pack	32
6. Tasking Issues	33
Scheduling Algorithm for Tasking	33
Mapping the Tasking Model onto an Existing Runtime Environment	34
Code Sharing for Objects of the Same Task Type	35
Pre-elaboration of Tasks	36
Task Storage Allocation	37
Activation Order of Tasks	38
Task Activation (Execution Order)	39
Storage_Error During Task Creation/Activation	40
Task Termination	41
Rendezvous Optimizations	42
Scheduling Event Caused by Delay Statement	43

Table of Contents

Delay Resolution	44
The Type Time	45
Evaluation of Delay Expression or Entry Family Index	46
Selective Wait Alternatives	47
Delay Alternatives	47
Task Priorities	48
Task Scheduling on Multiple Processors	48
Priority, Effect on Activation	49
Scheduling Order of Tasks	49
Rendezvous Between Tasks Without Priorities	50
Order of Abortion	50
Mechanism for Abort Completion	51
Abort Completion	51
Shared Variables	52
Pragma Shared	52
Direct Execution of Interrupt Entry Calls	53
Priority of Interrupt Entry Calls	54
Restrictions on Terminate Alternative	55
Nested Rendezvous Within an Interrupt Handler	56
7. Numeric Issues	57
Numeric_Error	57
Raising Numeric_Error Exceptions for an Operand in an Expression	58
Numeric_Error Exception for Real Operations	59
Comparisons of Real Operands	59
Numeric_Error Exception for Real Conversions	60
Rounding for Integer Conversions	61

Table of Contents

Accuracy of Static Real Expressions	62
Numeric_Error Exception for Nonstatic Universal Operations	63
Numeric_Error	64
Code Optimization	65
Integer Representation	66
Short_Integer and Long_Integer Representations	66
Float Representation	67
Short_Float and Long_Float Representations	67
8. Order Dependence	68
Effect of Incorrect Order Dependencies	68
Evaluation of Default Expressions	69
Range Constraint Evaluation	70
Index Evaluation Order	71
Component Subtype Elaboration Order	72
Index Constraints	73
Discriminant Checks for Incomplete Types	74
Discriminant Evaluation Order	76
Order of Elaboration Checks and Parameter Evaluation	77
Evaluation of an Indexed Component	78
Evaluation of a Slice	79
Evaluation Order of Component Expressions in a Record Aggregate	80
Evaluation Order of Component Associations in an Array Aggregate	81
Order of Constraint Checking	82
Evaluation of Operands in an Expressions	83
Assignment Statement Evaluation	84
Order of Evaluation of Parameter Associations	86

Table of Contents

Order of Parameter Copy-Back	87
Guard Condition Evaluation	88
Elaboration Order of Compilation Units	89
Program_Error	90
Elaboration of Generic Instantiations	91
9. Erroneous Program Issues	92
Effect of Erroneous Execution	92
Evaluation of Scalar Variables	93
Program_Error	94
Accessing Unchecked Deallocated Objects	95
10. Exception Issues	96
Exception Representation	96
Implementation-Defined Exceptions	97
Exception Handler Overhead	98
Non-Ada Exceptions	99
Main Program Termination	99
Code Motion	100
Pragma Suppress	101
Implementation-Defined Exceptions	101
11. Input-Output Issues	102
Form Parameter	102
External File Status	103
Input-Output of Access Type Objects	103
Concurrent Sharing of External Files	104
Null Form Argument String	104
File I/O Not Supported, Exceptions for	105

Table of Contents

Closing Temporary Files	106
Closing Sequential Files	107
Deleteion of Shared External Files	108
Name of an External File Associated with a Temporary File	109
Un-Interpretable Elements	110
Terminators	111
Waiting for Page Terminators	112
Sharing External Files	113
Buffering File Input and Output	114
End of Line and End of String	115
Representation of Non-Graphic Characters	116
Image of Non-Graphic Characters	118
12. Other Issues	119
Allowable Characters in Comments	119
Pragmas, Expressions Within	120
Implementation-Defined Attributes	121
Value of Scalar Out Parameters	122
Package Standard	123
Main Program Initiation	123
Main Program Parameters and Results	124
Elaboration Prior to Program Execution	124
Assignment Statement Constraint Checking	125
Shared Code for Generics	126
Expressions in Representation Clauses	126
Acceptance of Representation Clauses	127
Packing Algorithm for the Pragma Pack	127

Table of Contents

Additional Representation Pragmas	128
Expressions in Enumeration Representation Clauses	128
Limitations on Record Alignments	129
Ordering of Bits in Record Representation Clauses	129
Names of Implementation-Dependent Record Components	130
Expressions in Address Clauses	130
Overlays	131
Definition of Package System	131
Meaning of the Address Representation Attribute	132
Definition of Other Representation Attributes	132
Elaboration Checks for Interfaced Subprograms	133
Interface to Other Languages	133
Restrictions on Unchecked Type Conversions	134
Pragma Interface	134
Pragma Interface	135
Duration Representation	135
INDEX	136

Chapter 1

Preface

This catalogue describes those features of the Ada language that are allowed to vary among different implementations. It is derived from work done by the Ada RunTime Environment Working Group (ARTEWG), sponsored by ACM SIGAda, which produced the previous Catalogue of Ada Runtime Implementation Dependencies dated December, 1987. This previous version was based on analysis of the Ada Reference Manual (RM) and other associated documents, such as the Ada Compiler Validation Capability (ACVC) Implementors' Guide, and was structured following the same format as the RM. The current version has been restructured along functional similarities (issues) of the implementation features. New features have been added and approved Ada Issues¹ have been analyzed and are reflected in the catalogue where appropriate. The document has been cross-indexed to the RM and a document index added to make it more usable and complete. In addition, some small errors have been found and corrected.

The purpose of the document is to clearly identify those areas of the language that are implementation dependent and therefore are most likely to cause difficulties and limit the transportability of programs.

(KR) (—)

¹"Ada Issues" (AI) are identified for each question that arises about the interpretation of the standard. Each AI is assigned a unique number and all questions pertaining to that issue are addressed and catalogued with that Ada Issue by the language maintenance effort.

Chapter 2

Rationale

The Ada programming language was designed to support a wide variety of applications as well as a wide variety of computing systems. In an attempt to provide maximum language portability while still retaining requisite performance in that diverse environment, the Ada language definition had to walk the fine line between specifying everything completely and being flexible enough to permit efficient implementation on those diverse computers. That definition (ANSI/MIL-STD-1815A) provides considerable flexibility in the implementation of features which can impact the runtime performance of an Ada application.

For example, the implementer is free to choose the mechanism of parameter passing for composite types. This choice, and all the other choices the implementer makes, may have both positive and negative effects on an application program, especially in terms of its performance and its transportability. If the application has stringent requirements for either performance or transportability, then knowledge about the choices made in the various implementations will be useful.

Even though a close examination of the RM would uncover some of those areas, many of the features are subtle and require extensive review to identify. This catalogue is an attempt to itemize those "implementation dependencies" of Ada and describe their impact on the transportability and performance of Ada applications.

Thus, the main goal of this catalogue is to be the one place where all the areas of the RM which permit implementation flexibilities can be found. The RM explicitly calls these areas "implementation dependencies", hence the title of this catalogue.

There are several benefits that cataloguing these implementation dependencies provides. First, it serves as a guideline for analyzing Ada implementation technology. For each of these dependencies, Ada compilers can be examined to determine how the dependency is handled. Each application can then determine which implementation is best suited to the application.

Another benefit of the catalogue is that it provides a list of concerns for those who are procuring an Ada compiler. The Ada compiler buyer can examine the list to find the dependencies that are significant to their application. Those dependencies can then be among the criteria which the buyer can use in selecting an Ada compiler.

In analyzing the Ada language, all dependencies that were discovered were enumerated. Relying on knowledge about the implementation of some of these dependencies may be considered to be erroneous programming.

Catalogue of Ada Runtime Implementation Dependencies - Rationale

It should be noted that this catalogue is an evolutionary document. As official interpretations of the Ada language continue to be resolved, these often affect the flexibility which is allowed Ada implementers and must be reflected in the catalogue.

Chapter 3

Reader's Guide

To more accurately describe the relationship between implementation dependencies, this catalogue groups those dependencies that are logically related into each chapter. Each implementation dependency is further identified as to a particular RM chapter, section, subsection, and paragraph. To facilitate relating issues to the RM, each issue has been assigned a unique reference number with the following form:

`<chapter>.<section>[.<section>](<paragraph>).<sequence>`

Where "`<chapter>`" identifies the appropriate chapter of the RM, "`<section>[.<section>]`" identifies the relevant section (and subsection) of that chapter, "`(<paragraph>)`" identifies the paragraph number (shown in the margin of the RM), and "`<sequence>`" is a unique number assigned to identify multiple issues in the same paragraph.

In this catalogue, Ada runtime implementation dependency issues are shown using the following format:

Topic:

Question:

Reference:

Index Terms:

Rationale:

Example:

Each issue is identified by a brief topic line. The issue is then posed as a specific question. By identifying the answer to each question for a particular Ada implementation, an Ada application developer can gain a clearer understanding of the impact of those dependencies on the development of the particular application. The Reference line specifies the RM paragraph (as described earlier) followed by either "Explicit" or "Implicit". This indicates whether the listed issue is taken from an implementation dependency explicitly called out in the given paragraph of the RM, or whether it was inferred from the statements in that paragraph. The issue is related to general categories by entries in the "Index Terms" field. The index terms are followed by a short rationale describing why that issue is an Ada implementation dependency. Last, an Ada example is presented which further elucidates the implementation dependency.

Catalogue of Ada Runtime Implementation Dependencies - Guide

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Performance Issues

TOPIC: ELABORATION CHECKS (ELIMINATION OF)

Question: Does the implementation eliminate any runtime elaboration checks?

Reference: RM 3.9(4).1 - Implicit

Index Terms: *Elaboration, Runtime Checks, Optimization (Time)*

Rationale:

Elimination of elaboration checks at runtime may improve performance of an executing program.

Example:

```
package A is
  procedure ABC;
end A;

package body A is
  procedure ABC is
  begin
    null;
  end ABC;
end A;

with A;
procedure ABC_CALL is
begin
  A.ABC;
end ABC_CALL;
```

The elaboration rules guarantee a correct elaboration order for the above example. Therefore, a compiler could eliminate the elaboration check for the call A.ABC.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

Topic: STORAGE RECLAMATION (TIME OF)

Question: When is the storage reclamation performed?

Reference: RM 4.8(7).1 - Explicit

Index Terms: *Optimization (Time), Storage Management*

Rationale:

An implementation may (but need not) reclaim the storage occupied by an object created by an allocator, once this object has become inaccessible. This form of reclamation or garbage collection affects the speed and data storage of the executable program.

Example:

An implementation may use one of many garbage collection schemes to reclaim objects created by allocators which are no longer accessible. A garbage collection scheme usually works by periodically searching over all such objects to find the ones that are inaccessible. In addition to this overhead, these schemes require some ongoing bookkeeping to maintain the accessibility of objects created by allocators. This is a classic case of time versus space performance tradeoff.

The timing of the reclamation may be critical to the performance of the application. If the reclamation is performed too often, it may affect the timing; while if the reclamation is performed too seldom, the application runs the risk of running out of storage. Finally, being largely non-deterministic, the reclamation may occur during a critical portion of the application, affecting the timing of the application.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

Topic: EXCEPTIONS DURING STORAGE ALLOCATION

Question: When is the storage reclamation performed?

Reference: RM 4.8(13).1 and 4.8(5) - Implicit

Index Terms: *Optimization (Time), Storage Management*

Rationale:

AI-00397 states:

"When evaluating an allocator, a check is made that the designated object belongs to the allocator's designated subtype. CONSTRAINT_ERROR is raised if this check fails. This check can be made any time before evaluation of the allocator is complete. In particular, it is not defined whether this check is performed before creation of a designated object, evaluation of any default initialization expressions, or evaluation of any expressions contained in the allocator."

Depending on when the CONSTRAINT_ERROR is raised, a program may allocate more storage with one implementation than another, or execute additional functions within the initialization expressions.

Example:

```
type REC (D : INTEGER := FUNC_D; E : INTEGER := FUNC_E) is
  record
    C : INTEGER := FUNC_C;
  end record;
```

```
type AC_REC_3 is access REC(3, 3);
```

```
VAR1 : AC_REC_3 := new REC(4, 3);      -- is FUNC_C called?
VAR2 : AC_REC_3 := new REC'(IDENT_INT(4),
  FUNC);      -- is FUNC be called?
VAR3 : AC_REC_3 := new REC;            -- is FUNC_C called?
```

```
subtype INT_10 is INTEGER range 1..10;
type AC_INT_10 is access INT_10;
```

```
VAR4 : AC_INT_10 := new INTEGER'(11);  -- is storage allocated?
```

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: NULL STATEMENT (CODE GENERATED FOR)

Question: What code, if any, is generated for a **null** statement?

Reference: RM 5.1(5).1 - Implicit

Index Terms: *Optimization (Time and Space)*

Rationale:

The RM states that a **null** statement has no other effect than to pass to the next action. It is possible that some programming styles may generate a large number of **null** statements, with the assumption that no runtime penalty is imposed. This may not be a valid assumption for some compilers.

Example:

Given:

```
if <condition> then
  null;
else
  <sequence_of_statements>
end if;
```

The compiler could generate a branch instruction as part of the **null** statement, or the if statement could be restructured as:

```
if not <condition> then
  <sequence_of_statements>
end if;
```

Also, the code generated for:

```
      A := 4;
< <TARGET> > null;
      B := 5;
```

may not be the same as for:

```
      A := 4;
< <TARGET> > B := 5;
```

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: CASE STATEMENT IMPLEMENTATION

Question: If the compiler chooses between a jump table and a series of comparisons with branches to implement the case statement, what criteria does the compiler use to make the selection and what user control is supplied over the selection?

Reference: RM 5.4(1).1 - Implicit

Index Terms: *Optimization (Time and Space)*

Rationale:

The optimal choice may not always be determinable by the compiler. Applications which are limited in the amount of space available to store jump tables, but which have sufficient instruction space, may want to force the compiler to never use jump tables.

Example:

The above rationale points out a situation where the jump tables must be stored in data memory, which is separate from the instruction space. A more typical problem is selecting the break-off point from one method to the other when the case alternatives span a range that is large. For example:

```
SELECTOR : INTEGER;  
  
case SELECTOR is  
  when 1    => ...  
  ...  
  ...      -- 50 cases here  
  ...  
  when 750  => ...  
  when others => ...  
end case;
```

Case Statement Implementation (Continued)

Additional alternatives include a test/branch algorithm, a very large sparse branch table, or a hashing algorithm. Since memory space is not the only factor, trying to make a tradeoff on time versus space might be difficult for a compiler to do, even with **pragma OPTIMIZE**. Since the Ada language supports an **elsif** construct with which one could force the compiler to do a comparison/branch for each alternative, it might be reasonable for an implementation to ALWAYS use a jump table for the case statement regardless of the size of the table. Normally, even when a jump table is used, the lower and upper bounds of the case alternatives would restrict the size of the table to what will fit in memory, but it is very likely that a situation such as:

```
SELECTOR : LONG_INTEGER;  
  
case SELECTOR is  
  when INTEGER'FIRST => ...  
  
  ...  
  
  when INTEGER'LAST  => ...  
  when others        => ...  
end case;
```

will cause a compiler diagnostic indicating a capacity problem for the target computer.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: **ELABORATION OF BLOCK STATEMENTS**

Question: What runtime overhead is associated with the elaboration of a block statement?

Reference: RM 5.6(4).1 - Implicit

Index Terms: *Elaboration, Optimization (Time)*

Rationale:

Machines that use based addressing modes may require reallocation of base registers for accessing variables declared within the block. Furthermore, some implementations might have additional overhead to support exceptions raised within the block.

Example:

```
procedure BLOCK_TEST is
  T : INTEGER := 0;
begin
  NESTED_BLOCK:
  declare
    X : INTEGER;      -- local variable declared
  begin
    X := T;
  exception           -- special exception handler
    when CONSTRAINT_ERROR =>
      null;
  end NESTED_BLOCK;
end BLOCK_TEST;
```

Use of a local block limits the scope of operations, and therefore can improve maintenance of a program. However, if the overhead associated with the block construct is excessive, it may not be suitable for all real-time applications.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: STORAGE ALLOCATION FOR OBJECTS (STATIC)

Question: If a procedure is not called reentrantly, can local objects be allocated statically?

Section: RM 6.1(10).1 - Implicit

Index Terms: *Optimization (Time), Static Allocation*

Rationale:

The RM states that all subprograms can be called recursively and are reentrant. This question deals only with those cases in which sufficient information is available to the compiler to ensure that there is no requirement for more than one copy of local objects. In these instances, static allocation of local objects is possible, and static allocation may incur less runtime overhead than dynamic allocation.

If the amount of static memory available to an application is limited, dynamic allocation may be preferable to static allocation. Therefore, the user could desire the capability to choose whether static or dynamic allocation is used for local objects.

Example:

Assuming that it is known that the following function is not called reentrantly, the variable SUM may be allocated statically in the following instance:

```
function DOT_PRODUCT (LEFT, RIGHT : VECTOR) return REAL is
  SUM : REAL := 0.0;
begin
  ...
end DOT_PRODUCT;
```

TOPIC: PARAMETER PASSING MECHANISM

Question: For composite types passed as parameters, when is copy-in/copy-back used instead of "call by reference"?

Reference: RM 6.2(7).1 - Explicit

Index Terms: *Optimization (Time and Space), Subprogram Parameters*

Rationale:

The choice of the parameter passing mechanism may have a significant effect upon program performance. Using copy or reference to pass composite types may cause an application to incur an unacceptably large penalty in either execution time or storage usage.

Example:

In the following code fragment, passing the array parameter by copy may result in excessive runtime overhead. Furthermore, it may precipitate the exception `STORAGE_ERROR`.

```
CALL_COUNT : NATURAL := 0;
type GLOBAL_ARRAY_TYPE is array (1..100_000) of INTEGER;
GLOBAL_ARRAY : GLOBAL_ARRAY_TYPE;
...

procedure RECURSE_ARRAY (AN_ARRAY : out GLOBAL_ARRAY_TYPE) is
  LOWER,
  UPPER : POSITIVE;
begin
  -- Recursively initialize the first 100,000 elements
  -- of an array, 100 elements at a time.

  if CALL_COUNT < 100 then
    CALL_COUNT := CALL_COUNT + 1;
    LOWER := (CALL_COUNT - 1) * 100;
    UPPER := LOWER + 99;
    AN_ARRAY (LOWER..UPPER) :=
      (LOWER..UPPER => CALL_COUNT);
    RECURSE_ARRAY (GLOBAL_ARRAY);
  end if;
end RECURSE_ARRAY;
...

RECURSE_ARRAY (GLOBAL_ARRAY);
```

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: **PARAMETER PASSING CONVENTIONS**

Question: What storage mechanisms are used to pass parameters: registers, the stack, a special area reserved for parameters, or in some other storage mechanism?

Reference: RM 6.2(7).2 - Implicit

Index Terms: *Optimization (Time and Space), Subprogram Parameters*

Rationale:

The conventions for passing parameters may affect the execution efficiency of a program and, therefore, should be known in addition to the parameter passing mechanism.

Example:

An implementation may choose to pass parameters using the calling stack, registers, or some other means. The choice may depend upon the type and size of the parameter.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: PRAGMA INLINE

Question: Under what circumstances are subprograms expanded inline when pragma INLINE is specified?

Reference: RM 6.3.2(4).1 - Explicit

Index Terms: *Optimization (Time and Space), Subprogram Invocation*

Rationale:

The use of **pragma INLINE** recommends that the specified subprogram be expanded inline whenever it is called. The RM, though, specifically allows each compiler implementation an arbitrary decision as to whether a subprogram will actually be expanded inline. Thus, the rules that govern where **pragma INLINE** will be honored are important to application developers.

This is further complicated by AI-00200 which states that:

"If inline inclusion of a subprogram call is achieved due to **pragma INLINE**, an implementation is allowed to create a dependence of the calling unit on the subprogram body; when such a dependence exists, the unit containing the call is obsolete if the subprogram body is obsolete. Such dependences can be created even when the subprogram is created as a result of a generic instantiation."

Example:

```
procedure USE_INLINE is

  function INNER_FLOAT return FLOAT;
  function INNER_INTEGER return INTEGER is separate;
  pragma INLINE (INNER_FLOAT, INNER_INTEGER);

  function INNER_FLOAT return FLOAT is
    LOCAL : FLOAT;
  begin
    ...
    return LOCAL - FLOAT (INNER_INTEGER));
  end INNER_FLOAT;

begin
  ...
  if INNER_FLOAT = FLOAT (INNER_INTEGER) then
    ...
  end USE_INLINE;
```

An implementation may choose to "inline" only the function **INNER_FLOAT** that is completely declared within the compilation unit.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: **EXCEPTION RAISE OVERHEAD**

Question: What is the overhead associated with the execution of a raise statement?

Reference: RM 11.3(3).1 - Implicit

Index Terms: *Optimization (Time), Exceptions, Subprogram Parameters*

Rationale:

The overhead associated with raising an exception might influence an implementation choice between using the raise or returning a status as a parameter.

Example:

A user might design a package to do matrix manipulations. Included in this package could be a subprogram that finds the inverse of a matrix. If raising an exception is a very high overhead operation, one might define the subprogram as follows:

```
procedure INVERSE (IN_MATRIX : in MATRIX;  
                   OUT_MATRIX : out MATRIX;  
                   MATRIX_IS_SINGULAR : out BOOLEAN);
```

But if raising an exception is not a high overhead operation, a better interface might be:

```
IS_SINGULAR : exception;  
function INVERSE (IN_MATRIX : in MATRIX) return MATRIX;
```

where the function would raise the exception IS_SINGULAR if the matrix was singular.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: EXCEPTION HANDLER OVERHEAD

Question: What is the overhead associated with selecting a handler when an exception is raised?

Reference: RM 11.4(1).1 - Implicit

Index Types: *Optimization (Time), Exceptions*

Rationale:

The overhead associated with selecting an exception handler when an exception is raised might influence implementation choices of Ada implementations that are used for an application.

Example:

Two techniques have been identified for locating and selecting an appropriate exception handler: dynamic tracking and static mapping. Static mapping requires a search to locate the appropriate set of handlers for selection, while dynamic tracking does not. But, dynamic tracking requires a stack of exception handlers to be maintained at runtime, while static mapping does not. The choice of techniques in Ada implementations may influence the performance of an application.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: ERROR DETECTION, EFFICIENCY

Question: Could use of the optimizer cause exceptions that otherwise might not occur?

Reference: RM 11.6(5).1 - Explicit

Index Terms: *Optimization (Time), Runtime Checking*

Rationale:

Through code motion and reordering of expression evaluation an exception that would otherwise not occur might be raised. During debugging the code might be tested without the optimizations turned on, and then after the code was thoroughly tested the code would be recompiled with full optimizations. The user would not want new exceptions introduced at this point without some warning that this might occur.

Example:

Given that X, Y and Z are floating point variables, consider the following example:

```
X := FLOAT_SAFE_LARGE;  
Y := FLOAT_SAFE_LARGE;  
Z := X / 2.0 + Y / 2.0;
```

If the optimizer were to rearrange the expression to be $(X + Y)/2.0$, the addition could result in an overflow that would not otherwise occur.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: PRAGMA SUPPRESS

Question: For each check that can be suppressed, is there an overhead associated with suppressing the check?

Reference: RM 11.7(18).2 - Implicit

Index Terms: *Optimizations (Time), Exceptions, Runtime Checking*

Rationale:

When exceptions are suppressed, the user may not want extra code added to explicitly ignore exceptions. Furthermore, if the exception is handled through hardware, it may not be possible to fully suppress the exception.

Example:

Consider a machine that has an unmaskable interrupt associated with fixed point overflow. An Ada implementation for this machine might associate an interrupt routine for the overflow interrupt that raised `NUMERIC_ERROR` (SEE AI-00387). In order to suppress `OVERFLOW_CHECK` on a particular object, the implementation might generate an inline exception handler for all arithmetic operations on that object that could cause overflow.

TOPIC: PRAGMA SUPPRESS

Question: Under what circumstances is the `SUPPRESS` pragma ignored?

Reference: RM 11.7(20).1 - Explicit

Index Terms: *Optimization, Exceptions, Runtime Checking*

Rationale:

If a program includes pragma `SUPPRESS`, the user needs to know whether or not the given check will be omitted, since this can affect program performance.

Example:

The same example as above is applicable here.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: MACHINE CODE INSERTS

Question: Is the package `MACHINE_CODE` supported and how are the machine features supported?

Reference: RM 13.8(4-6).1 - Explicit

Index Terms: *Optimization (Time)*

Rationale:

This package is one way for expressing machine level operations that are not otherwise expressible in Ada language; the other way is an interface to an assembly language subprogram.

Example:

Example machine level operations are those which change the machine state, such as disabling interrupts or manipulating machine registers. An implementation may require the pragma `INLINE` for any machine code subprogram or may exclude parameter passing.

TOPIC: TEMPORARY FILE STATUS

Question: What is the status of temporary files after completion of a main program?

Reference: RM 14.2.1(3).2 - Implicit

Index Terms: *Optimization (Space), Input-Output*

Rationale:

The status of temporary files following the completion of the main program may potentially affect subsequent program execution due to the unavailability of file storage. The RM states that temporary files are not accessible following completion of the main program but does not require the deletion of the associated external files (RM 14.1(7)).

Example:

A program that makes excessive use of temporary files without explicitly deleting them prior to completion may disrupt execution of a subsequent program unless the implementation ceases the existence of the external files upon program completion.

Catalogue of Ada Runtime Implementation Dependencies - Performance Issues

TOPIC: PRAGMA INLINE

Question: Under what circumstances are subprograms, with **pragma INLINE**, not expanded inline? Conversely, under what circumstances are subprograms, without **pragma INLINE**, expanded inline?

Reference: RM Annex B(4).1 - Explicit

Index Terms: *Optimization (Time)*

Rationale:

The RM does not require that an implementation must expand the subprogram call inline. An implementation is free to follow or ignore the recommendation of the pragma. In fact the absence of the pragma does not ensure against inline expansion.

Example:

An implementation may limit the ability to expand a subprogram inline to those subprograms whose bodies are not separate from their specifications.

TOPIC: PRAGMA SUPPRESS

Question: What exactly is the effect of the **pragma SUPPRESS**?

Reference: RM Annex B(14).1 - Explicit

Index Terms: *Optimization (Time), Runtime Checking*

Rationale:

The presence of a **SUPPRESS** pragma gives permission to an implementation to omit certain runtime checks. The pragma does not issue an order to suppress the checks; therefore an implementation can choose not to suppress the checks.

Example:

An implementation may choose not to suppress some checks despite the presence of the pragma when the checks are implemented by the underlying hardware. It would either be impossible to suppress or extremely inefficient to do so.

Chapter 5

Storage Issues

TOPIC: REPRESENTATION OF A LITERAL

Question: How are literals represented in the executable program?

Reference: RM 4.2(1).1 - Implicit

Index Terms: *Optimization (Time and Space), Static Allocation*

Rationale:

Literal values may be represented in the executable program as storage data or as part of the actual machine instructions in the data field of an immediate machine instruction. Additionally, redundant occurrences of the same literal in the same or different scope may be represented by the same bit string value in storage. The choice made will affect the size of code and data storage of the program.

Example:

Some literal values, especially small ones such as integers, boolean, and characters, may be represented as a value in the data field within an immediate instruction of the target computer. Example instructions may be "store immediate" or "add immediate". An implementation may, for efficiency reasons, choose to represent these values directly in the data field of these immediate instructions. Thus the integer expression:

... A + 2 ...

may be encoded in the following machine instruction sequence:

```
LOAD A
ADD 1220 -- where 1220 is the storage location
          -- containing the integer value 2
```

or it may be encoded in the following alternative instruction sequence:

```
LOAD A
ADDI 2   -- that is, "add immediate" the small integer value 2.
```

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: **AGGREGATES (REPRESENTATION OF)**

Question: How are aggregates represented in the executable program?

Reference: RM 4.3(1).1 - Implicit

Index Terms: *Optimization (Space)*

Rationale:

Aggregate values may be represented in the executable program as contiguous storage data or they may be encoded within a sequence of statements. The choice made will affect the size of the code and data storage of the program.

Example:

Aggregates may be represented as a contiguous block of storage data. This is true when all the values of the aggregate's components are known at compile time. Thus the aggregate:

... (1|4|7 => 15, 2|3|5|6 => -5, 8..10 => 0) ...

may be represented in a block of ten storage units containing the values -5, 0, and 15. Conversely, aggregates may be encoded within a sequence of code. This is necessary for aggregates with component values determined at runtime. The sequence of code generated usually varies with the use of the aggregate. An assignment statement provides an example of this implementation choice:

B := (1..10 => 5, 11..100 => 0);

The implementation in this case may choose to generate a sequence of code equivalent to the following Ada statements, instead of generating a contiguous block of storage:

```
for I in 1..10 loop
  B (I) := 5;
end loop;
for I in 11..100 loop
  B (I) := 0;
end loop;
```

Other elements that affect the choice made by the implementation are the availability of space, the availability of high level machine instructions, such as block moves and compares, and the percentage of unique values in the aggregate.

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: **LIBRARY UNIT (UNREFERENCED ELEMENTS OF)**

Question: Are unneeded elements of a package body included in the executable program?

Reference: RM 10.4(1).1 - Implicit

Index Terms: *Optimization (Space)*

Rationale:

Including unneeded library units or runtime routines in an executable program results in unneeded storage space being used. This affects the storage requirements of application programs.

Example:

If all subprograms in a package body are linked into a program instead of just those needed, storage space is consumed that is not necessary. For example, suppose a subprogram does no input or output, but uses the INTEGER_IO.PUT procedure to do integer-to-character conversion. Will all the rest of TEXT_IO be included in the executable program?

```
with TEXT_IO; use TEXT_IO;
procedure NO_OUTPUT is

  package INT_IO is new INTEGER_IO (INTEGER); use INT_IO;
  ...

begin
  ...

  PUT (TO => A_STRING, ITEM => AN_INTEGER);
  ...

end NO_OUTPUT;
```

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: SIZE SPECIFICATION FOR COMPOSITE TYPES

Question: What impact does the length clause have on the packing algorithm of composite types?

Reference: RM 13.2(5).1 - Explicit

Index Terms: *Representation Clauses, Optimization (Space), Storage*

Rationale:

The RM specifies that a size specification for a composite type may affect the size of the gaps between the storage areas allocated to consecutive components. An implementation may allow the user to specify a size (i.e., length) which may pack the components of that type.

Example:

Consider an array type A with 200 integer components where each component may have a size of three bits. An implementation may allow the length of A to be 1600 bits, implying that each component is packed one to a byte; or A to be 800 bits, implying that two components are packed to a byte; or even 600 bits, implying no storage space between components.

TOPIC: STORAGE RESERVED FOR A COLLECTION SIZE

Question: What is considered to be part of the storage reserved for a collection of an access type?

Reference: RM 13.2(8).1 - Implicit

Index Terms: *Storage, Representation Clause*

Rationale:

Contents of the storage reserved for a collection of an access type is implementation dependent. The control afforded by the length clause is relative to the implementation conventions. To accurately predict the storage requirements for a collection, the user must know the manner of allocation used for access collections.

Example:

The allocation scheme used by an implementation may use internal tables and links which are allocated from the storage space for the access collection. The formulas for calculating these must be known for accurate prediction of collection sizes.

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: STORAGE RESERVED FOR A TASK ACTIVATION

Question: What is considered to be part of the storage reserved for the activation of a task?

Reference: RM 13.2(10).1 - Implicit

Index Terms: *Tasking, Storage, Representation Clauses*

Rationale:

The contents of the storage reserved for an activation of a task is implementation dependent. The control afforded by the length clause is relative to the implementation conventions. To accurately predict the storage requirements for an activation of a task, the user must know the manner of allocation used for task activations.

Example:

The task management scheme of an implementation uses internal tables and links (sometimes called task control blocks) which may be included in the storage space for the activation of a task. The formulas for calculating these must be known for accurate prediction of task activation sizes.

TOPIC: EXPRESSIONS IN ENUMERATION REPRESENTATION CLAUSES

Question: What are the limitations on expressions that appear in enumeration representation clauses?

Reference: RM 13.3(1).1 - Implicit

Index Terms: *Representation Clauses*

Rationale:

The RM defines that the interpretation of the expressions that appear in representation clauses is implementation dependent. These interpretations are documented in Appendix F of the RM.

Example:

An implementation may have a convention that only allows unsigned integers to represent enumeration literals.

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: LIMITATIONS ON RECORD ALIGNMENTS

Question: What are the restrictions on the allowable alignments for record representation clauses?

Reference: RM 13.4(4).1 - Explicit

Index Terms: *Representation Clauses*

Rationale:

The RM defines that an implementation is permitted to place restrictions on the allowable alignments for record representation clauses. These restrictions are documented in Appendix F of the RM.

Example:

Due to hardware restrictions an implementation may require that all record representation clauses have an alignment that places them on word boundaries.

TOPIC: OVERLAPPING STORAGE BOUNDARIES IN RECORD REPRESENTATION CLAUSES

Question: Is a component allowed to overlap a storage boundary within record representation clauses?

Reference: RM 13.4(5).2 - Explicit

Index Terms: *Storage, Representation Clauses, Optimization (Space)*

Rationale:

The RM permits an implementation to define whether and how a component of a record is allowed to overlap a storage boundary. Permitting a component to overlap a storage boundary will support tighter packing of components in the storage layout of a record.

Example:

Addressing constraints of some computer systems may force implementations to prohibit boundary crossings.

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: STORAGE LAYOUT FOR RECORD TYPES

Question: Where does an implementation place a record component which has no component clause in a record representation clause?

Reference: RM 13.4(6).1 - Explicit

Index Terms: *Storage, Representation Clauses*

Rationale:

If no component clause is given for a component, the choice of the storage place for the component is left to the implementation. The implementation may unintentionally place a component in a gap that the user had left open for logical reasons.

Example:

One implementation may place all components without component clauses following the placement of the last user-placed component, while another implementation may attempt to place components in unused gaps.

TOPIC: RUNTIME OVERLAYS

Question: Does the implementation provide pragmas for the specification of program overlays?

Reference: RM 13.5(10).1 - Explicit

Index Terms: *Storage, Overlay*

Rationale:

Many non-Ada programs rely on the use of program overlays to achieve execution when storage capacity is limited. Reuse of these programs by an Ada program requires that some form of overlay facility exist in order to support execution under similar storage capacity limitations.

Example:

An application might have a start-up subprogram, `START_UP`, that is never used after it is executed the first time. It makes sense to allow `START_UP` to be overlayed with another subprogram, `FOO_BAR`, that is needed later during the execution of the application. An implementation pragma might specify this overlay as follows:

```
pragma OVERLAY (START_UP, FOO_BAR);
```

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: **STORAGE LAYOUT FOR ARRAY TYPES**

Question: What is the order in which elements of multidimensional arrays are stored?

Reference: RM 13.9(6).1 - Implicit

Index Terms: *Storage, Array Layout*

Rationale:

The RM does not state whether arrays should be in row-order or column-order, and the difference would not normally affect the user. However, because of the likely reuse of FORTRAN subroutines, this could be important with the **pragma INTERFACE**. Many compilers are used for non-numeric processing and follow the Algol/Pascal tradition of treating the first index position as most rapidly varying. FORTRAN implementations are required by the language to make the last index vary the most rapidly.

Example:

Consider the following array declaration:

FLOAT_ARRAY : array (1..10, 1..20) of FLOAT;

The array requires 200 objects of type FLOAT, but may be stored as 10 rows each of 20 objects, or as 20 columns each of 10 objects.

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: **SIZE OF A DIRECT ACCESS FILE**

Question: What is the maximum size of a direct access file?

Reference: RM 14.2.1(3).1 - Explicit

Index Terms: *Input-Output, Storage (File Size)*

Rationale:

Dependence upon a specific maximum size for a direct access file for input and output processing may potentially affect the reusability, and transportability of a program unit.

Example:

A program unit that depends upon the maximum size of a direct access file equal or greater than the value of `INTEGER'LAST` may not be reusable under an implementation that supports a smaller value. The following code fragment illustrates the dependency:

```
function RANDOM return POSITIVE_COUNT;  
...  
SET_INDEX (FILE, RANDOM);  
-- CONSTRAINT_ERROR may be raised.
```

Catalogue of Ada Runtime Implementation Dependencies - Storage Issues

TOPIC: PRAGMA OPTIMIZE

Question: What exactly is the effect of the pragma OPTIMIZE or conversely, the effect of not using the pragma at all?

Reference: RM Annex B(8).1 - Explicit

Index Terms: *Optimization (Time and Space)*

Rationale:

The RM is not specific about the semantics of this pragma. It only says that TIME or SPACE will be the primary optimization criterion. Furthermore, the lack of this pragma may produce still another criterion for optimization.

Example:

For each feature of the language or combination of features there may be several "best" transformations, some that are space efficient while others that are time efficient. This could result in the unusual situation where forcing the compiler to select TIME over SPACE may result in a less efficient overall program.

Not specifying the pragma may result in yet another transformational strategy which could combine some of both TIME or SPACE efficient optimizations.

TOPIC: PRAGMA PACK

Question: What is the effect of the pragma PACK on a type in an Ada program?

Reference: RM Annex B(9).1 - Explicit

Index Terms: *Optimization (Time and Storage)*

Rationale:

The RM only requires that this pragma specify that storage minimization should be the main criterion when selecting the representation of a record or array type. The specific details are left up to an implementation.

Example:

Consider a pair of components that are three bits and six bits in length on a byte addressable machine. Will the second component be allocated at the fourth bit of the first byte or the first bit of the second byte? The algorithm to implement PACK determines its utility on a large number of applications.

Chapter 6

Tasking Issues

TOPIC: SCHEDULING ALGORITHM FOR TASKING

Question: What is the scheduling algorithm that determines the execution order of tasks?

Reference: RM 9(2).1 - Implicit

Index Terms: *Tasking, Optimization (Time)*

Rationale:

The result of the program may depend on the scheduling algorithm. In a monoprocessor environment, the execution of parallel tasks can be interleaved. Possible interleaving algorithms include: pre-emptive priority, time slice, and run-till-blocked.

Example:

Several tasks producing output may have different results depending on the scheduling algorithm.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **MAPPING THE TASKING MODEL ONTO AN EXISTING RUNTIME ENVIRONMENT**

Question: How does the Ada model for tasking map onto the existing runtime environment of the target computer system?

Reference: RM 9(2).2 - Implicit

Index Terms: *Tasking*

Rationale:

The developer of an Ada compilation system must decide how the Ada model for tasking will be represented on the existing runtime environment of the target computer system. The target computer system, which may or may not have an executive, has constraints on how computer processes are managed. If those constraints are a subset of the Ada model for tasking, the developer may decide to use only the existing runtime environment primitives to manage Ada tasks. Otherwise, the developer will choose to use a subset of those existing runtime environment primitives that conforms to the Ada model for tasking. The result of this decision affects all aspects of Ada tasking, including the creation, activation, scheduling, termination, and abortion of tasks. All of these affect the behavior of the application.

Example:

The developers of an Ada compilation system whose target computer system has a UNIX² executive did not map Ada tasks onto UNIX processes, because the UNIX process model is not a subset of the Ada model for tasking. Instead, Ada programs are mapped onto a single UNIX process. As a result, if a single Ada task is blocked for an I/O operation, it will suspend the execution of the whole Ada program.

²Unix is a trademark of Bell Laboratories

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **CODE SHARING FOR OBJECTS OF THE SAME TASK TYPE**

Question: Is the code of a task body shared among multiple occurrences of the same task type?

Reference: RM 9.1(6).1 - Implicit

Index Terms: *Tasking, Optimization (Space)*

Rationale:

Significant memory savings can be realized if the code of the body of a task type can be shared. Yet this must be assessed in situations which may not be appropriate for code sharing, such as distributed computer systems with non-shared memory, computers with segmented memory, or applications which must be intensively debugged.

Example:

In a loosely coupled distributed system, task objects of the same task type that are allocated to different processors would require a separate local copy of the task body. In an architecture like the extended memory MIL-STD-1750A, task objects of the same task type that are allocated to different memory segments would require their own copy of the task body for each memory segment. Finally, when an application with task objects of the same task type is to be intensively debugged, the user might require a copy of the task body for each task object if unique debugging requests, such as breakpoints, are to be requested.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **PRE-ELABORATION OF TASKS**

Question: Can tasks in library level packages be statically allocated?

Reference: RM 9.2(2).1 - Implicit

Index Terms: *Tasking, Static Allocation, Optimization (Time), Elaboration*

Rationale:

For many programs, the number of tasks is known at compile time. These tasks are declared in library level packages. They execute as infinite loops and never terminate. In such systems, the job of the system designer is easier if tasks can be statically allocated at compile/link-time. Static allocation permits ROMable code, reduces runtime overhead, and has the advantage that the memory requirements of code and space are known at compile/link-time.

Example:

In some applications, the time from power-on reset to full operational status may have a requirement under 1 millisecond. If an application has several tasks, it may be impossible to elaborate/activate them in sufficient time during program initialization. If most of the elaboration can be done at link time, it will reduce the start up overhead substantially.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **TASK STORAGE ALLOCATION**

Question: How is storage for tasks allocated?

Reference: RM 9.2(2).2 - Implicit

Index Terms: *Tasking, Optimization (Time and Space)*

Rationale:

When task objects are created dynamically, storage must be allocated for them. There may be different mechanisms used for this allocation. Additionally, there may be mechanisms for deallocation of storage for tasks that have terminated. Also, there may be limits to the total amount of storage available for task objects. All of this may affect the efficiency and portability of a program that creates many task objects.

Example:

Suppose a recursive subprogram creates many task objects at each execution. Storage might be deallocated immediately upon termination of these tasks, or upon return to the caller of the subprogram that created the task objects, or delayed until the main subprogram is completed. Depending on the timing of the program, the storage allocated to tasks may exceed the storage capacity of the computer system.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **ACTIVATION ORDER OF TASKS**

Question: In what order are tasks activated?

Reference: RM 9.3(1).1 - Explicit

Index Terms: *Tasking, Priorities, Order Dependence*

Rationale:

When several tasks are activated in parallel, the order of their elaboration may affect program execution.

Example:

In the following example, the string STR in the task TASK_1 will have a different length depending on the order of elaboration:

```
...
procedure X is
  GLOBAL : POSITIVE := 1;

  task type T1;

  TASK_1 : T1;
  TASK_2 : T1;
  TASK_3 : T1;

  function SIDE_EFFECT return POSITIVE is
  begin
    GLOBAL := GLOBAL + 1;
    return GLOBAL;
  end SIDE_EFFECT;

  task body T1 is
    STR : STRING (1..SIDE_EFFECT);
  begin
    ...

  end T1;

begin
  ...

end X;
```

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **TASK ACTIVATION (EXECUTION ORDER)**

Question: Can a task, following its activation but prior to the completion of activation of tasks declared in the same declarative part, continue execution?

Reference: RM 9.3(2).1 - Implicit

Index Terms: *Tasking, Priorities, Order Dependence*

Rationale:

The activation of tasks proceeds in parallel. Correct execution of a program may depend on a task continuing execution after its activation is completed but before all other tasks activated in parallel have completed their respective activations.

Example:

The following code fragment illustrates that the result of program execution will change depending upon whether all parallel task activation must be completed prior to continued execution.

```
declare
  task SERVER is
    entry E; pragma PRIORITY (SYSTEM.PRIORITY'LAST);
  end SERVER;

  task REQUESTOR;

  function CALL_SERVER return BOOLEAN is
  begin
    select
      SERVER.E;
      return TRUE;
    else
      return FALSE;
    end select;
  end CALL_SERVER;

  task body SERVER is
  begin
    accept E;
  end SERVER;

  task body REQUESTOR is
    SERVER_EXECUTING : BOOLEAN := CALL_SERVER;
  begin
    TEXT_IO.PUT_LINE (BOOLEAN'IMAGE (SERVER_EXECUTING));
  end REQUESTOR;
begin ... end;
```

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **STORAGE_ERROR DURING TASK CREATION/ACTIVATION**

Question: If the allocation of a task object raises the exception `STORAGE_ERROR`, when is the exception raised?

Reference: RM 9.3(3).1 - Implicit

Index Terms: *Tasking, Storage, Exceptions*

Rationale:

The RM does not explicitly define when `STORAGE_ERROR` must be raised should a task object exceed the storage allocation of its creator or master. The exception must be raised no later than task activation; however, an implementation may choose to raise it earlier, (e.g., when the task is declared).

Example:

The following code fragment illustrates that program execution will change depending upon when the exception is raised. In the illustration it is assumed that the code body of `STORAGE_ERROR_TASK` exceeds the allocation of the task enclosing the block.

```
...  
  
declare  
  
  task STORAGE_ERROR_TASK;  
  task body STORAGE_ERROR_TASK is  
  
    -- Propagation of STORAGE_ERROR will occur  
    -- prior to activation of INNER_TASK if raised  
    -- at task creation.  
  
    ...  
  
    package body INNER is  
      task body INNER_TASK ...  
  
    ...  
  
  end INNER;  
  
  -- Propagation of STORAGE_ERROR will occur after  
  -- INNER_TASK is activated if raised at task activation.  
  
begin  
...
```

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **TASK TERMINATION**

Question: When the main program terminates (abnormally), what happens to tasks that depend on library packages?

Reference: RM 9.4(13).1 - Explicit

Index Terms: *Tasking*

Rationale:

NOTE: THIS IMPLEMENTATION DEPENDENCY HAS BEEN UPDATED BY AI-00399:

"If a task depends on a library package, then after (normal) termination of the main (sub)program, the environment task must continue to wait for all library tasks to terminate; if all library tasks terminate, then the program as a whole terminates.

If execution of the main program is abandoned, either because the main program raises an exception or because an exception is raised by the elaboration of a library unit, the effect on executing library tasks is undefined. In particular, the tasks can be aborted or they can continue to run until normal termination.

The RM does not define whether such tasks are required to terminate. For some applications it is desirable that such tasks not terminate. Other applications may desire that all tasks terminate when the main program terminates."

Example:

The main program may contain initialization code only, with the library level tasks performing all the major processing. The main program terminates when the initialization is done, leaving the library tasks to continue. However if the main program terminates due to an exception propagating out of the main program thread, the library tasks may be aborted, or left to run indefinitely, depending on the implementation.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: RENDEZVOUS OPTIMIZATIONS

Question: What rendezvous optimizations are available?

Reference: RM 9.5(14).1 - Implicit

Index Terms: *Tasking, Optimization (Time), Priorities*

Rationale:

The efficiency of the tasking operations is a concern for many applications. Various implementations of rendezvous have been proposed and documented. Some implementations of tasking rendezvous may be more appropriate than others for certain applications. Real-time applications that use tasking should be concerned about the rendezvous implementation and the possible tasking special cases that can be optimized by the compiler and runtime environment.

Example:

A simplified measurement of the efficiency of a rendezvous is the number of context swaps required to complete the rendezvous. Most general purpose rendezvous implementations require two context swaps; one to switch to the called task, and one at the end of the accept body to switch back to the calling task. If the compiler can recognize or be informed of special conditions, then the rendezvous overhead can be reduced. One such special condition is a passive server task that is executed only when called. A rendezvous call to a passive server task (or monitor task) can be optimized to execute in the context of the caller. This means that no context swaps may be required. Because the passive server task can execute in the context of the caller, it does not need its own stack or task control block.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **SCHEDULING EVENT CAUSED BY DELAY STATEMENT**

Question: What is the minimum value in a delay expression that causes a scheduling event?

Reference: RM 9.6(1).1 - Implicit

Index Terms: *Tasking, Delays, Priorities*

Rationale:

The **delay** statement suspends further execution of the task that executes the **delay** statement, for at least the duration specified by the value of the expression in the **delay** statement. There is an overhead associated with requesting a delay from the runtime system. This overhead may exceed the requested delay period. Unless the requesting task has been aborted, the runtime system may return control directly to that task or schedule another task for execution.

Example:

The statement **delay (0.0)** has a delay value less than any overhead associated with the delay request to the runtime system. Unless that task has been aborted, the runtime system may return control to the requesting task or schedule another task for execution.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **DELAY RESOLUTION**

Question: Is the execution accuracy of delay statements consistent with SYSTEM.TICK and DURATION'SMALL?

Reference: RM 9.6(4).1 - Implicit

Index Terms: *Tasking, Delays*

Rationale:

For real-time applications, it is important that the delay statement be executed with an accuracy consistent with the values specified for SYSTEM.TICK and DURATION'SMALL. In particular, the delay statement should be executed with an accuracy no less than SYSTEM.TICK.

Example:

The following code fragment illustrates that unless the accuracy is consistent with SYSTEM.TICK and DURATION'SMALL, the result of program execution may be affected. In the example, the timed entry call may be reduced to a conditional entry call when SYSTEM.TICK is less than DURATION'SMALL.

```
select
  SERVER.E;
...

or
  delay DURATION (SYSTEM.TICK) - DURATION'SMALL;
...

end select;
```

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: THE TYPE TIME

Question: What is the implementation overhead associated with the type TIME?

Reference: RM 9.6(5).1 - Implicit

Index Terms: *Optimization (Time), Delays*

Rationale:

The implementation overhead associated with type TIME may be significant in developing real-time applications that have critical timing requirements. Furthermore, this overhead should not prevent changes in the value of TIME from being resolved at the accuracy of SYSTEM.TICK.

Example:

The following code fragment illustrates that program execution may depend upon this overhead.

```
START_TIME := CLOCK + OVERHEAD_FACTOR;
while CLOCK - START_TIME < DURATION (SYSTEM.TICK) loop
-- Process data for this tick
...
end loop;
```

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: EVALUATION OF DELAY EXPRESSION OR ENTRY FAMILY INDEX

Question: When are the expressions of an open delay alternative or the entry family index in an open accept alternative evaluated?

Reference: RM 9.7.1(5).2 - Implicit

Index Terms: *Tasking, Order Dependence, Optimization (Time), Delays, Priorities*

Rationale:

After all the open alternatives have been determined, the expressions in the open delay alternative or the entry family index in an open alternative are evaluated in the process of selecting an open alternative for execution. It is up to the implementation to determine whether all those expressions must be evaluated before a selection is made. The choice affects both side effects and the performance behavior of the select statement.

Example:

Two examples illustrate the point. In both cases, when there is a caller on the entry queue for the entry the value of X may be 0 or 1 depending on the implementation choice.

```
X : INTEGER;
function FOO return INTEGER is
...
begin
...
  X := 1;
...
end FOO;

-- example with delay expression
X := 0;
select
  accept E;
or
  delay (DURATION (FOO));
end select;

-- example with entry family index
X := 0;
select
  accept E;
or
  accept F (FOO);
end select;
```

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: SELECTIVE WAIT ALTERNATIVES

Question: What is the algorithm to select from the open alternatives in a selective wait?

Reference: RM 9.7.1(6).1 - Explicit

Index Terms: *Tasking, Order Dependence*

Rationale:

The RM states that the choice is arbitrary. However, some applications may be concerned with how this is implemented.

Example:

For some applications a choice of open alternatives based on the longest waiting time is desired. Other applications may require that if tasks of different priorities are queued on different open alternatives, the alternative that has the highest priority task waiting on it be selected.

TOPIC: DELAY ALTERNATIVES

Question: What algorithm is used to select from delay alternatives of the same delay in a selective wait?

Reference: RM 9.7.1(8).1 - Explicit

Index Terms: *Tasking, Delays*

Rationale:

The RM states that the choice is arbitrary. However, some applications may be concerned with how this is implemented.

Example:

```
select
  accept A;
or
  delay 5.0;
  UPDATE_SOME_GLOBAL;
or
  delay 5.0;
end select;
```

If there are no callers for A in the delay period, the procedure call to UPDATE_SOME_GLOBAL may or may not be called depending on the algorithm used to select between the delay alternatives.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: TASK PRIORITIES

Question: If priorities are supported, what are their implementation characteristics?

Reference: RM 9.8(1).1 - Explicit

Index Terms: *Tasking, Priorities*

Rationale:

Applications have different execution requirements. Time and response critical applications must be guaranteed to run when required. Knowing whether or not priorities are implemented, the range of priorities available, and the default priority of tasks and the main program is necessary for designing time-critical and response-critical systems.

Example:

If a system designer is aware that a particular runtime environment supports three levels of priority (subtype PRIORITY is INTEGER range 0..2) and that the assigned default priority is the lowest priority (0), the following conclusions may be made: tasks that respond to interrupt handlers should be priority 2, foreground processing should be priority 1 and background processing should be priority 0. This particular setup exemplifies a disadvantageous situation of having a single priority level available for foreground processing. Since some foreground processes will be more time-critical than others, elaborate scheduling will be needed to meet requirements. This scheduling will increase the complexity of the designed code.

TOPIC: TASK SCHEDULING ON MULTIPLE PROCESSORS

Question: Is preemptive scheduling supported for tasks distributed over multiple processors?

Section: RM 9.8(4).1 - Implicit

Index Terms: *Tasking, Priorities*

Rationale:

An Ada program may be distributed over more than one processor. The developer needs to know if priorities are supported system wide or by processor.

Example:

For tasks that are running on separate physical processors, it is possible for a medium priority task to be blocked on PROCESSOR-01 (by a higher priority task on PROCESSOR-01) while a lower priority task on PROCESSOR-02 is executing.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: PRIORITY, EFFECT ON ACTIVATION

Question: What priority is used during task activation?

Section: RM 9.8(4).2 - Implicit

Index Terms: *Tasking, Priorities, Activation*

Rationale:

AI-00288 provides a NON-BINDING (Ada'83 implementations need not support it) interpretation of which priority the activation of a task occurs at:

"A task activation should be performed with the priority of the task being activated or the priority of the task causing the activation, whichever is higher."

Since this is non-binding, current implementations may simply activate the task at its specified priority.

Example:

If a low priority task is activated by a high priority task, some implementations will suspend the high priority task while a medium priority task is allowed to execute. This may result in timing problems for an application.

TOPIC: SCHEDULING ORDER OF TASKS

Question: What is the scheduling order of tasks with the same priority or without explicit priorities?

Reference: RM 9.8(5).1 - Explicit

Index Terms: *Tasking, Order Dependence*

Rationale:

This must be known by the designers of embedded systems to ensure required system performance.

Example:

Assume an Ada program has three tasks of the same priority. The three tasks execute delay statements such that they will all be eligible to execute at the exact same time. If the tasks do not execute in FIFO manner, this may affect program execution.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: RENDEZVOUS BETWEEN TASKS WITHOUT PRIORITIES

Question: What is the priority of a rendezvous between two tasks without explicit priorities?

Reference: RM 9.8(5).2 - Explicit

Index Terms: *Tasking, Priorities*

Rationale:

This information may be necessary to ensure required program performance with respect to other tasks that have defined priorities.

Example:

Two tasks without explicit priority conduct a rendezvous. If the priority given to the rendezvous is higher than a task with an explicit priority, the program may execute in an unexpected manner.

TOPIC: ORDER OF ABORTION

Question: What is the order of abortion for tasks specified in an abort statement?

Reference: RM 9.10(4).1 - Explicit

Index Terms: *Tasking, Priorities, Abort, Order Dependence*

Rationale:

The results of a program may vary depending on the order of abortion of its tasks.

Example:

Assume there exist three tasks A, B, and C running within an Ada program and the following statement is executed:

abort A, B, C;

If the user assumes that the tasks are aborted in a left to right order, but the implementation chooses not to follow such an order, the program execution may be different.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: MECHANISM FOR ABORT COMPLETION

Question: When does a task which has been aborted become completed?

Reference: RM 9.10(6).1 - Explicit

Index Terms: *Tasking, Abort*

Rationale:

When a task has been aborted, it may become completed at any point from the time the abort statement is executed until its next synchronization point. Depending on when an implementation actually causes the task to complete, the results of an aborted task may be different.

Example:

Suppose a task is updating a variable that is visible to other tasks, prior to a synchronization point. If the task is aborted just prior to the update, it may leave the variable unchanged if it becomes completed immediately, or it may update the variable and then become completed at the synchronization point. This could affect the results of the whole program.

TOPIC: ABORT COMPLETION

Question: What are the results if a task is aborted while updating a variable?

Reference: RM 9.10(8).1 - Explicit

Index Terms: *Tasking, Abort*

Rationale:

An implementation may defer completion of a task if it is aborted while updating a variable, and thus prevent a variable from being undefined. This may be crucial in the case of a common variable.

Example:

Two tasks both reference a variable A. The first task uses A for calculations, the other task updates A periodically. If the second task is aborted during the update of A, then A may become undefined. This might raise an exception in the first task.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: SHARED VARIABLES

Question: In the absence of the SHARED pragma, are local copies of shared variables used within tasks?

Reference: RM 9.11(8).1 - Implicit

Index Terms: *Tasking, Erroneous Execution*

Rationale:

If no local copies are used, erroneous programs will operate correctly on one implementation and not on others.

Example:

Many compilation systems generate code that makes use of temporary local copies of objects, usually in registers. However, some compilation systems may update objects only as direct, indivisible operations, or may recognize an object as being shared and treat it as if pragma SHARED had been used. In these cases, an erroneous program would execute correctly until the compilation system was altered. If the program was transported to a different compilation system, errors could occur which are likely to be difficult to isolate.

TOPIC: PRAGMA SHARED

Question: How is pragma SHARED implemented for the supported types?

Reference: RM 9.11(11).1 - Implicit

Index Terms: *Tasking, Optimization (Time and Space)*

Rationale:

The performance of programs using the SHARED pragma may vary because of the implementation approach for this pragma.

Example:

Some architectures may directly provide a full range of indivisible operations in hardware for scalars and access types. In other cases, the indivisible operations may be implemented by disabling interrupts, performing the operation, and restoring interrupts. In these cases, the overhead associated with interrupt control may be substantial.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **DIRECT EXECUTION OF INTERRUPT ENTRY CALLS**

Question: Under what conditions, if any, will an implementation enable the hardware to directly execute the accept statements associated with an interrupt entry call?

Reference: RM 13.5.1(2).1 - Implicit

Index Terms: *Representation Clauses, Tasking*

Rationale:

The RM defines that an interrupt entry call acts as an entry call issued by a hardware task whose priority is higher than the priority of the main program and any user-defined task. This enables an implementation to implement the interrupt entry call having the hardware directly execute the appropriate accept statements, which is absolutely necessary for many real-time applications.

Example:

One condition that is required on many implementation for hardware direct execution of interrupt entry calls is that there are no other calls to the entry from the application software. An implementation may require a special pragma that specifies that only hardware interrupts will cause interrupt entry calls.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: PRIORITY OF INTERRUPT ENTRY CALLS

Question: What assumptions can be made regarding the suspension of interrupt handlers following completion of the **accept** body corresponding to an interrupt entry call?

Reference: RM 13.5.1(2).2 - Implicit

Index Terms: *Representation Clauses, Tasking*

Rationale:

The RM stipulates that interrupt entry calls act as a call from a task with a priority that is higher than any user-defined priority, resulting in the corresponding **accept** body to execute at this priority. This implies that, unless the interrupt handler task is explicitly assigned the highest user-defined priority, it may be suspended following completion of the **accept** body. Many implementations may choose not to suspend the task in order to optimize interrupt handlers. This is possible because the "loop" statement below can be eliminated on some implementations.

Example:

Consider the following interrupt handler:

```
task INTERRUPT_HANDLER is
  entry INTERRUPT_ENTRY;
  for INTERRUPT_ENTRY use at 16#40#;
end INTERRUPT_HANDLER;

task body INTERRUPT_HANDLER is
begin
  loop
    accept INTERRUPT_ENTRY do
      ...
    end INTERRUPT_ENTRY;
  end loop;
  -- task may be suspended
end INTERRUPT_HANDLER;
```

An implicit assumption that the implementation will only suspend the above task upon reaching the **accept** is erroneous.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: RESTRICTIONS ON TERMINATE ALTERNATIVE

Question: What further requirements are imposed by an implementation for selection of the **terminate** alternative that may appear in the same **select** statement with an **accept** alternative for an interrupt **entry**?

Reference: RM 13.5.1(3).1 - Explicit

Index Terms: *Tasking*

Rationale:

Further requirements may be necessary to prevent the premature selection of the **terminate** alternative.

Example:

Selecting the **terminate** alternative may complete the task which contains the only **accept** statements which can handle the interrupt **entry** calls, leaving the hardware unserved. This would be fatal to most applications.

Catalogue of Ada Runtime Implementation Dependencies - Tasking Issues

TOPIC: **NESTED RENDEZVOUS WITHIN AN INTERRUPT HANDLER**

Question: What is the effect of an interrupt handler attempting to perform a rendezvous?

Reference: RM 13.5.1(5).1 - Implicit

Index Terms: *Tasking, Representation Clauses*

Rationale:

Depending upon the machine, code that is directly executed from an interrupt vector, or similar mechanism, may cause the machine to be in a state that prevents a nested rendezvous to proceed.

Example:

Consider the following interrupt handler:

```
task INTERRUPT_HANDLER is
  entry INTERRUPT_ENTRY;
  for INTERRUPT_ENTRY use at 16#40#;
end INTERRUPT_HANDLER;

task body INTERRUPT_HANDLER is
begin
  accept INTERRUPT_ENTRY do
    SOME_TASK.SOME_ENTRY;    -- Erroneous
  end INTERRUPT_ENTRY;
end INTERRUPT_HANDLER;
```

An implementation may choose to restrict the actions performed within an accept body of an interrupt entry.

Chapter 7

Numeric Issues

TOPIC: **NUMERIC_ERROR**

Question: Does the implementation raise **NUMERIC_ERROR** on an intermediate operation when the larger expression can be correctly computed?

Reference: RM 3.5.4(10).1 - Explicit

Index Terms: *Numerics, Runtime Checks*

Rationale:

An implementation is not required to raise **NUMERIC_ERROR** (SEE AI-00387) for an intermediate value if the larger expression can produce the correct result.

Example:

```
A : INTEGER := INTEGER'LAST;
```

```
...
```

```
A := (A + 1) - 2;
```

The intermediate value $(A + 1)$ is greater than **INTEGER'LAST** but the result of the entire expression is valid.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: **RAISING NUMERIC_ERROR EXCEPTIONS FOR AN OPERAND IN AN EXPRESSION**

Question: In an expression, if an operand yields a value outside the base type, yet the result is mathematically correct, is NUMERIC_ERROR exception raised?

Reference: RM 4.5(7).1 - Implicit

Index Terms: *Numerics, Runtime Checks*

Rationale:

The NUMERIC_ERROR (SEE AI-00387) exception check for the value of an expression is required to be performed on the final result of the expression. For some base types, an implementation may calculate intermediate results with greater precision than that requested for the base type of the expression, which may avoid the NUMERIC_ERROR exception as long as the result is a value of the base type.

Example:

Consider:

$$X := (A * B) / C;$$

Suppose A, B, C, and X are variables of the same floating point type. NUMERIC_ERROR need not be raised if $A * B$ lies outside the range of the safe numbers for the base type as long as the quotient lies within the range. If the quotient does not lie within the range, NUMERIC_ERROR should be raised.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: NUMERIC_ERROR EXCEPTION FOR REAL OPERATIONS

Question: What happens when the result of a real operation is not in the range of safe numbers?

Reference: RM 4.5.7(7).1 - Explicit

Index Terms: *Numerics, Runtime Checks, Exceptions*

Rationale:

The RM does not require an implementation whose MACHINE_OVERFLOW is FALSE to raise NUMERIC_ERROR (SEE AI-00387) when the result of a real operation has a model interval that is undefined. This error condition may go undetected on computers that do not permit easy detection of overflow situations.

Example:

When the computer hardware cannot detect an overflow condition, the implementation is not obligated to insert additional code to determine if the resulting value is out of the range of safe numbers for a real type operation.

TOPIC: COMPARISONS OF REAL OPERANDS

Question: For a relational operator what is the range of accuracy for the relational comparison?

Reference: RM 4.5.7(10).1 - Implicit

Index Terms: *Numerics*

Rationale:

For the result of a relation between two real operands, the result can be any value obtained by applying the mathematical comparison to a value arbitrarily chosen in the corresponding operand model intervals. This potential imprecision makes the results of comparisons implementation dependent.

Example:

Consider that not $(A < B)$ need not be equal to $(A \geq B)$ under certain circumstances. This happens when the model intervals associated with A and B have more than one value in common. The values of A and B are, for non-model numbers, implementation-defined within certain limitations, which consequently makes the results of comparisons implementation dependent.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: **NUMERIC_ERROR EXCEPTION FOR REAL CONVERSIONS**

Question: What happens when the result of a conversion to a real type is not in the range of safe numbers?

Reference: RM 4.6(7).1 - Implicit

Index Terms: *Numerics, Runtime Checks, Exceptions*

Rationale:

The RM does not require an implementation whose MACHINE_OVERFLOW is FALSE to raise a NUMERIC_ERROR (SEE AI-00387) when the result of a conversion has a model interval that is undefined. This error condition may go undetected on computers that do not permit easy detection of overflow situations.

Example:

When the computer hardware cannot detect an overflow condition, the implementation is not obligated to insert additional code to determine if the resulting value is out of the range of safe numbers for a real type operation.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: **ROUNDING FOR INTEGER CONVERSIONS**

Question: In a conversion of a real value to an integer type, if the operand is halfway between two integers which way is it rounded?

Reference: RM 4.6(7).2 - Explicit

Index Terms: *Numerics*

Rationale:

The RM does not specify what kind of rounding is to be performed when converting real values whose values lie in between two integers to integer types.

Example:

Here is a list of some of the different kinds of rounding that can be applied to real-to-integer type conversion:

INTEGER (1.5) -> 2 and INTEGER (-1.5) -> -2 -- normally expected

or

INTEGER (1.5) -> 2 and INTEGER (2.5) -> 2 -- round to even

or

INTEGER (-1.5) -> -1 and INTEGER (-2.5) -> -3 -- round to even (2's complement)

or

INTEGER (1.5) -> 2 and INTEGER (-1.5) -> -1 -- round up

or

INTEGER (1.5) -> 1 and INTEGER (-1.5) -> -2 -- round down

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: **ACCURACY OF STATIC REAL EXPRESSIONS**

Question: What differences exist between the accuracy of compile-time evaluation versus runtime evaluation of static expressions?

Reference: RM 4.9(12).1 - Implicit

Index Terms: *Numerics*

Rationale:

An implementation is not required to produce the same value for a static expression by compile-time evaluation on a host computer system as it would produce for the same expression at runtime on the target computer system. These values are only required to be in the same model number interval. An implementation is required to evaluate a static expression at compile-time when the context requires a static expression. The consequence of this potential imprecision makes the results of comparisons implementation-dependent.

Example:

Consider the following declaration with the static real expression:

```
SOMETHING : REAL := EXPX + EXPY + EXPZ;
```

Now consider this function:

```
function FUNX (A : REAL := EXPX,  
              B : REAL := EXPY,  
              C : REAL := EXPZ) return REAL;
```

If within FUNX there is a relational expression:

```
... (A + B + C) = SOMETHING ...
```

it may not produce an equal comparison when FUNX is called with no parameters. This happens when the model intervals associated with the REAL type have more than one value in common. The values of SOMETHING and (A + B + C) are, for non-model numbers, implementation-defined within certain limitations, which consequently makes the results of comparisons implementation dependent.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: **NUMERIC_ERROR EXCEPTION FOR NONSTATIC UNIVERSAL OPERATIONS**

Question: What happens if the result of the operation is a real value whose absolute value exceeds the largest safe number of the most accurate predefined floating point or an integer value greater than `SYSTEM.MAX_INT` or less than `SYSTEM.MIN_INT`?

Reference: RM 4.10(5).1 - Implicit

Index Terms: *Numerics, Runtime Checks, Exceptions*

Rationale:

The RM does not require an implementation to raise a `NUMERIC_ERROR` (SEE AI-00387) if the result of the operation is a real value whose absolute value exceeds the largest safe number of the most accurate predefined floating point or an integer value greater than `SYSTEM.MAX_INT` or less than `SYSTEM.MIN_INT`. This error condition may go undetected on computers that do not permit easy detection of overflow situations.

Example:

```
value1 := 10_000;  
value2 := 30_000;  
value3 := value1 * value2;
```

The above computation may, or may not raise `NUMERIC_ERROR` on an implementation with `MAX_INT` of 32767.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: **NUMERIC_ERROR**

Question: Is the raising of NUMERIC_ERROR supported in hardware?

Reference: RM 11.1(6).1 - Implicit

Index Terms: *Numerics, Runtime Checks*

Rationale:

Unless OVERFLOW_CHECK is suppressed, there potentially could be a runtime check for overflow for each arithmetic expression. This runtime expense can be reduced with the use of hardware checks. (See also 4.5.7(7).1 for more discussion on NUMERIC_ERROR.)

NOTE: AI-00387 specifies in a NON-BINDING interpretation: "Wherever the Standard requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR should be raised instead."

Since this interpretation is non-binding, implementations are free to raise either NUMERIC_ERROR or CONSTRAINT_ERROR wherever NUMERIC_ERROR could be implicitly raised.

Example:

Hardware support for the raising of NUMERIC_ERROR might be supplied by associating an interrupt handler with floating point and fixed point overflow interrupts. This interrupt handler could then cause the runtime system to raise the exception. If no such interrupt exists, or if it is not used to raise the exception, an implementation might generate code to check for overflow before or after every arithmetic operation.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: CODE OPTIMIZATION

Question: Does the use of a type that has a range wider than that of the base type ever suppress the raising of an exception?

Reference: RM 11.6(6).1 - Explicit

Index Terms: *Numerics, Exceptions, Optimization (Time)*

Rationale:

For expression evaluation, an implementation is allowed to use a type that has a range wider than that of the base type of the operands, provided that this delivers the exact result. Use of such a wider type might allow an operation that might otherwise raise `NUMERIC_ERROR` (SEE AI-00387).

Example:

Consider the following example:

```
function MAY_RAISE_NUMERIC_ERROR return BOOLEAN is
  A, B : POSITIVE := POSITIVE'LAST;

begin
  return (A + 1) < (B + 1);
end MAY_RAISE_NUMERIC_ERROR;
```

A call to this function may raise `NUMERIC_ERROR` since the evaluation of `A + 1` results in a number outside the range of the positive numbers.

TOPIC: INTEGER REPRESENTATION

Question: What is the representation of the predefined type INTEGER?

Reference: RM Annex C(6).1 - Implicit

Index Terms: *Numerics*

Rationale:

An implementation is free to choose the range of values for the predefined type INTEGER.

Example:

An implementation usually chooses the range of the normal integer representation for a machine as the range for the predefined INTEGER type. For a 16-bit machine this may be the range of 16-bit signed integers. In this instance, the following code fragment is potentially non-reusable depending upon the time of day it is executed:

```
use CALENDAR;  
...  
  
INTEGER_SECONDS := INTEGER (SECONDS (CLOCK));  
-- NUMERIC_ERROR (see AI-00387) may be raised.
```

TOPIC: SHORT_INTEGER AND LONG_INTEGER REPRESENTATIONS

Question: What is the representation of the predefined type SHORT_INTEGER and LONG_INTEGER?

Reference: RM Annex C(7).1 - Implicit

Index Terms: *Numerics*

Rationale:

An implementation may provide additional predefined integer types.

Example:

An implementation usually chooses the range of the short integer representation for a machine as the range for the predefined SHORT_INTEGER type and the range of the double length integer representation for a machine as the range for the predefined LONG_INTEGER type. For a 16-bit machine this may be the range of 8-bit signed integers for SHORT_INTEGER and 16-bit signed integers for LONG_INTEGER.

Catalogue of Ada Runtime Implementation Dependencies - Numeric Issues

TOPIC: **FLOAT REPRESENTATION**

Question: What is the representation of the predefined type FLOAT?

Reference: RM Annex C(9).1 - Implicit

Index Terms: *Numerics*

Rationale:

An implementation is free to choose the range and accuracy of values for the predefined type FLOAT.

Example:

An implementation usually chooses the range and accuracy of the normal floating point representation for a machine as the range for the predefined FLOAT type. For a 32-bit machine this may be the range of 32-bit signed floating point representation.

TOPIC: **SHORT_FLOAT AND LONG_FLOAT REPRESENTATIONS**

Question: What is the representation of the predefined type SHORT_FLOAT and LONG_FLOAT?

Reference: RM Annex C(10).1 - Implicit

Index Terms: *Numerics*

Rationale:

An implementation may provide additional predefined floating point types.

Example:

An implementation usually chooses the range and accuracy of the short floating point representation for a machine as the range for the predefined SHORT_FLOAT type and the range and accuracy of the double length floating point representation for a machine as the range for the predefined LONG_FLOAT type. For a 32-bit machine this may be the range and accuracy of 16-bit signed floating point representation for SHORT_FLOAT and the range and accuracy of 64-bit signed floating point representation for LONG_FLOAT.

Chapter 8

Order Dependencies

TOPIC: EFFECT OF INCORRECT ORDER DEPENDENCIES

Question: What is the effect of incorrect order dependencies?

Reference: RM 1.6(9).1 - Explicit

Index Terms: *Order Dependence*

Rationale:

The user may wish to know the effect of incorrect order dependencies (e.g., does the compiler raise `PROGRAM_ERROR`). This question applies for all specific instances indicated as questions in other chapters.

Example:

```
S, T : STRING (1..4);
G   : POSITIVE := 1;

function F return POSITIVE is
begin
  G := G + 1;
  return G - 1;
end F;
...

S (F..F) := T (F..F);
```

Depending on the order of evaluation, the assignment statement may result in raising `CONSTRAINT_ERROR`, or different slices may be selected in legal slice assignments.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: EVALUATION OF DEFAULT EXPRESSIONS

Question: In what order are default expressions for components of a record evaluated?

Reference: RM 3.2.1(15).1 - Explicit

Index Terms: *Order Dependence, Numerics*

Rationale:

The results of default initialization of record components may vary depending on the order of evaluation of the expressions.

Example:

Assume:

```
G : INTEGER := 0;  
function F return INTEGER is  
begin  
  G := G + 1;  
  return G - 1;  
end F;  
...
```

And:

```
type DANGEROUS_RECORD is  
record  
  A: INTEGER := F;  
  B: INTEGER := F;  
end record;
```

DR: DANGEROUS_RECORD;

DR.A and DR.B have the values 0 and 1, or 1 and 0, respectively, depending on the order of evaluation of the initialization expressions.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: RANGE CONSTRAINT EVALUATION

Question: What is the order of evaluation of simple expressions specifying range bounds?

Reference: RM 3.5(5).1 - Explicit

Index Terms: *Order Dependence, Numerics*

Rationale:

The RM permits an implementation to evaluate the simple expressions of a range constraint in any order. As a result, a program may execute correctly on some implementations but not on others.

Example:

If a simple expression contains a function with side effects, the order of evaluation of the range constraint may produce different results. Consider the following:

```
G : INTEGER := 0;
function F return INTEGER is
begin
  G := G + 1;
  return G - 1;
end F;
...

subtype S is INTEGER range F..F;
```

The range of subtype S is either "0..1" or "1..0" (i.e., null) depending upon the order of evaluation of F.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: INDEX EVALUATION ORDER

Question: What is the order of evaluation for ranges of index constraints?

Reference: RM 3.6(10).1 - Explicit

Index Terms: *Order Dependence, Numerics*

Rationale:

Varying the order of evaluation of index constraint ranges may produce different effects.

Example:

If a simple expression contains a function with side effects, the evaluation of the index constraint range may produce different results. Consider the following:

```
G : INTEGER := 0;
function F return INTEGER is
begin
  G := G + 1;
  return G - 1;
end F;
...
S : array (F..F) of INTEGER;
```

The value of the range F..F may be either 0..1 or 1..0, depending on the evaluation order.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **COMPONENT SUBTYPE ELABORATION ORDER**

Question: What is the order of the elaboration of component subtype indications for an array?

Reference: RM 3.6(10).2 - Explicit

Index Terms: *Order Dependence, Elaboration*

Rationale:

Varying the order of elaboration of component subtype indications may produce different effects.

Example:

If a simple expression contains a function with side effects, the elaboration of component subtype indications may produce different results. Consider the following:

```
G : INTEGER := 0;  
function F return INTEGER is  
begin  
  G := G + 1;  
  return G - 1;  
end F;  
...
```

```
S : array (1..10) of INTEGER range F..F;
```

The value of the range F..F may be either 0..1 or 1..0, depending on the elaboration order.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: INDEX CONSTRAINTS

Question: In what order are index constraints evaluated?

Reference: RM 3.6.1(11).1 - Explicit

Index Terms: *Order Dependence, Numerics*

Rationale:

Index constraints may be different if the order of evaluation of constraint limits is changed.

Example:

If a simple expression contains a function with side effects, the evaluation of the index constraint may produce different results. Consider the following:

```
G : INTEGER := 0;
type ABC is array (INTEGER range < >) of INTEGER;

function F return INTEGER is
begin
  G := G + 1;
  return G - 1;
end F;
...

S : ABC (F..F);
```

The index constraint of S is either "0..1" or "1..0" (i.e., null) depending upon the order of evaluation of F.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: DISCRIMINANT CHECKS FOR INCOMPLETE TYPES

Question: When a discriminant constraint is applied to a private type before its full declaration or to an incomplete type before its full declaration, 3.3.2(8) and 3.7.2(5) require that if a discriminant is used to constrain a subcomponent, its value must be checked for compatibility with the subcomponent's type. However, prior to the full declaration, no subcomponents exist (other than the discriminants themselves). When is the required check performed?

Reference: RM 3.7.2(5).1 - Implicit

Index Terms: *Order Dependence, Exceptions*

Rationale:

AI-00007 indicates that two choices were considered in deciding when the deferred check must be performed:

- 1) perform a deferred check as soon as possible, i.e., when each component subtype definition using a discriminant is elaborated; and
- 2) perform a deferred check when the type being constrained is "completely" declared, i.e., when all subcomponents have been declared.

The recommended solution allows deferred checks to be performed in accordance with choice 1 and requires that deferred checks be performed no later than in accordance with choice 2. To see the differences between the choices and the consequences of the recommendation, consider the following:

Discriminant checks for incomplete types (continued)

Example:

```

package P is
  subtype Int7 is Integer range 1..7;
  subtype Int6 is Integer range 1..6;
  subtype Int5 is Integer range 1..5;

  type T_Int7 (D7 : Int7) is private;
  type T_Int6 (D6 : Int6) is private;
  type T_Int5 (D5 : Int5) is private;

  subtype T_Cons is T_Int7(6);      -- (1)
private
  type T_Int7 (D7 : Int7) is
    record
      C76 : T_Int6(D7);            -- (2)
    end record;
  type T_Int6 (D6 : Int6) is
    record
      C65 : T_Int5(D6);            -- (3)
      CF : T_Int5(Func);           -- (4)
    end record;
  type T_Int5 (D5 : Int5) is
    record
      CF : String (Func .. D5);    -- (5)
    end record;                  -- (6)
end P;

```

In this example, the full declaration of T_Int7 is not the complete declaration of T_Int7 because the declaration of T_Int6 is not yet complete. Similarly, T_Int6's full declaration is not the complete declaration for T_Int6 or for T_Int7 because of the use of T_Int5. Finally, T_Int5's full declaration is a complete declaration for T_Int5 and so completes the declaration of T_Int6 and T_Int7.

CONSTRAINT_ERROR would be raised at point (1) (in accordance with the first compatibility check) if the discriminant's value were not in the range 1..7. Since it is in the required range, no exception is raised. Subsequent checks are deferred compatibility checks that depend on the use of the discriminant in T_Int7's full type declaration. The declaration at (2) requires yet another deferred check, since T_Int6 has not yet been fully (or completely) declared, and similarly the declarations at (3) require deferred checks. At the end of T_Int5's declaration, T_Int5, T_Int6, and T_Int7 have been completely declared and no additional deferred checks are required. The recommendation requires that all deferred checks for types whose declaration is completed by T_Int5's declaration be performed no later than point (6). Since the value 6 is not compatible with D6's use at point 3, the checks deferred from point (2) and point (1) will require that CONSTRAINT_ERROR be raised. The exception can be raised at any point between point (3) (the first point at which a deferred check can fail) and point (6) (the point by which all deferred checks in this example must have been completed).

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **DISCRIMINANT EVALUATION ORDER**

Question: What is the order of evaluation of discriminant associations?

Reference: RM 3.7.2(13).1 - Explicit

Index Terms: *Order Dependence*

Rationale:

A discriminant constraint may vary based on the order of evaluation of the discriminant associations.

Example:

```
type A_REC (A_SIZE, B_SIZE : INTEGER) is
  record
    ...
  end record;

G : INTEGER := 100;
function F return INTEGER is
begin
  G := G + 1;
  return G;
end F;
...

S : A_REC (F, F);
```

The values for A_SIZE and B_SIZE in record S are either 101 and 102, or 102 and 101, respectively, depending upon the order of evaluation of F.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **ORDER OF ELABORATION CHECKS AND PARAMETER EVALUATION**

Question: Is the check that the body of a subprogram has been elaborated made before or after the parameters of a call have been evaluated? The corresponding question arises for generic instantiations.

Reference: RM 3.9(5).1 - Implicit

Index Terms: *Elaboration, Runtime Checks, Order Dependence*

Rationale:

AI-00406 states:

"It is not defined whether the check that the body of a subprogram has been elaborated is made before or after the actual parameters of a call have been evaluated.

Similarly, it is not defined whether the check that the body of a generic unit has been elaborated is made before or after the generic actual parameters of an instantiation have been evaluated."

Example:

If a procedure is called with a function as an actual (or default) parameter, and the procedure body has not yet been elaborated, an implementation may execute the procedure prior to raising `PROGRAM_ERROR`. This may change the behavior of the program, although the program is clearly in error.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: EVALUATION OF AN INDEXED COMPONENT

Question: In what order are the expressions in an indexed component evaluated?

Reference: RM 4.1.1(4).1 - Explicit

Index Terms: *Order Dependence*

Rationale:

The side effects of the expression evaluation may be different based on the order of evaluation chosen. The expressions found in a prefix may be evaluated before the expressions for the indices of the indexed component or vice versa. Likewise, the expressions within the prefix or the expressions for the indices may be evaluated in either left-to-right or right-to-left order.

Example:

The evaluation of an indexed component evaluates the index expressions and the prefix in an order that is not defined by the language. Since some order is chosen, this means that if any evaluable construct in a prefix is evaluated, then all such constructs must be evaluated before evaluating the index expressions, or vice versa. The following example illustrates this point:

```
type A_ARRAY is array (1..20, 1..20) of INTEGER;  
function FUNCX (X,Y : INTEGER) return A_ARRAY;  
  
... FUNCX (F1, F2) (F3, F4) ...
```

Assuming that F1, F2, F3, and F4 are functions with side effects, then the following evaluation orders are allowed:

```
F1, F2, F3, F4  
F3, F4, F1, F2  
F2, F1, F3, F4  
F3, F4, F2, F1  
F1, F2, F4, F3  
F4, F3, F1, F2  
F2, F1, F4, F3  
F4, F3, F2, F1
```

All other evaluation orders are not permitted.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: EVALUATION OF A SLICE

Question: In what order are the expressions in a slice evaluated?

Reference: RM 4.1.2(4).1 - Explicit

Index Terms: *Order Dependence*

Rationale:

The side effects of the expression evaluation may be different based on the order of evaluation chosen. The expressions found in a prefix may be evaluated before the expressions for the range of the slice or vice versa. Likewise, the expressions within the prefix or the expressions for the range of the slice may be evaluated in either left-to-right or right-to-left order.

Example:

The evaluation of a slice evaluates the expressions of the range and the prefix in an order that is not defined by the language. Since some order is chosen, this means that if any evaluable construct in a prefix is evaluated, then all such constructs must be evaluated before evaluating the range expressions of the slice, or vice versa. The following example illustrates this point:

```
type B_ARRAY is array (1..20) of INTEGER;  
function FUNCX (X, Y : INTEGER) return B_ARRAY;  
  
... FUNCX (F1, F2) (F3..F4) ...
```

Assuming that F1, F2, F3, and F4 are functions with side effects, then the following evaluation orders are allowed:

```
F1, F2, F3, F4  
F3, F4, F1, F2  
F2, F1, F3, F4  
F3, F4, F2, F1  
F1, F2, F4, F3  
F4, F3, F1, F2  
F2, F1, F4, F3  
F4, F3, F2, F1
```

All other evaluation orders are not permitted.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: EVALUATION ORDER OF COMPONENT EXPRESSIONS IN A RECORD AGGREGATE

Question: In what order are the expressions in the component associations of a record aggregate evaluated?

Reference: RM 4.3.1(3).1 - Explicit

Index Terms: *Order Dependence, Records*

Rationale:

AI-00189 clarifies this implementation dependency by stating:

"If an array aggregate or string literal is an expression in a record aggregate and a bound of the array aggregate or string literal is determined by the value of a discriminant, the expression giving the value of that discriminant is evaluated before the array aggregate or literal is evaluated; otherwise, the expressions given in the component associations of a record aggregate are evaluated in some order that is not defined by the language."

The value of the aggregate may be different based on the order of evaluation chosen.

Example:

In the evaluation of a record aggregate, the expressions in the component associations are evaluated in an order that is not defined by the language. As an example, for the record aggregate:

... (A => FUNX, B => FUNY, C => FUNZ) ...

the order of evaluation might be FUNX, FUNY, FUNZ or FUNZ, FUNY, FUNX, as well as others. If FUNX or FUNZ affect each other either directly through common global variables or indirectly through calls to the runtime system, the behavior or the results of those evaluations may be different.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **EVALUATION ORDER OF COMPONENT ASSOCIATIONS IN AN ARRAY AGGREGATE**

Question: In what order are the expressions for component associations in an array aggregate evaluated?

Reference: RM 4.3.2(10).1 - Explicit

Index Terms: *Order Dependence, Arrays*

Rationale:

The expressions in component associations may be evaluated in any arbitrary order. The value of the array aggregate may be different based on the order of evaluation chosen.

Example:

In the evaluation of an array aggregate, the expressions in the component associations are evaluated in an order that is not defined by the language. As an example for the array aggregate:

... (1..10 => FUNX, 11..100 => FUNY, 101..150 => FUNZ) ...

the order of evaluation might be FUNX, FUNY, FUNZ or FUNZ, FUNY, FUNX, as well as others. If FUNX or FUNZ affect each other, either directly through common global variables or indirectly through calls to the runtime system, the behavior or the results of those evaluations may be different.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **ORDER OF CONSTRAINT CHECKING**

Question: Are constraints on choices or on the values of expressions of component associations checked before all choices of expressions are evaluated?

Reference: RM 4.3.2(11).1 - Explicit

Index Terms: *Order Dependence, Runtime Checking*

Rationale:

The check that the value of an index in an array aggregate belongs to an index subtype can be made before or after all choices have been evaluated. Similarly, the check that a subcomponent value belongs to the subcomponent's subtype can be performed before or after all subcomponent expressions have been evaluated. If the consequence of performing a constraint check is to raise an exception, then an additional consequence is that certain expressions or choices may not yet have been evaluated.

Reference AI-00018.

Example:

Take the simple aggregate below:

... (1..5 => EXPX, 6..10 => EXPY) ...

Implementations are free to generate code that performs constraint checking either after one or both expressions are evaluated. Assume that the side effects from evaluating the expressions in any implementation will cause the value of EXPY to raise a `CONSTRAINT_ERROR` if it is evaluated before EXPX. If an implementation performs constraint checking after each expression is evaluated, then the EXPX will not be evaluated if the order of evaluation is EXPY, EXPX.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: EVALUATION OF OPERANDS IN AN EXPRESSION

Question: In what order are the operands in an expression evaluated?

Reference: RM 4.5(5).1 - Explicit

Index Terms: *Order Dependence, Numerics*

Rationale:

The value of the operand may be different based on the order of evaluation chosen. The operands found in expressions may be evaluated in any order.

Example:

In the evaluation of an expression, the operands in the expression are evaluated in an order that is not defined by the language. As an example, for the expression:

OPNDX + OPNDY + OPNDZ

the order of evaluation might be OPNDX, OPNDY, OPNDZ or OPNDZ, OPNDY, OPNDX, as well as others. If OPNDX or OPNDZ affect each other either directly through common global variables or indirectly through calls to the runtime system, the behavior or the results of those evaluations may be different.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: ASSIGNMENT STATEMENT EVALUATION

Question: In what order are the variable name and expression evaluated in an assignment statement?

Reference: RM 5.2(3).1, 5.2(4).1 - Explicit

Index Terms: *Order Dependence, Erroneous Execution*

Rationale:

If the evaluation of the variable name or the expression has side effects, the order of evaluation could affect the result.

Note that AI-00333 restricts the implementation flexibility somewhat:

"In an assignment, the variable name and the expression are evaluated in some order that is not defined by the language, except when the evaluation of the expression depends on the subtype of the variable and this subtype can only be determined by evaluating the variable; in this case, the variable is evaluated first. (Such a situation can occur only when the expression is a string literal or an array aggregate having positional associations or an others choice, since the applicable index constraint in these cases is given by the subtype of the variable.)

Example:

```
procedure ASSIGN is
  A : array (1..4) of INTEGER;
  I : INTEGER := 4;

  function SIDE_EFFECT return INTEGER is
  begin
    I := 5;
    return 3;
  end SIDE_EFFECT;

begin
  -- ASSIGN
  A (I) := SIDE_EFFECT;
end ASSIGN;
```

The effect of this program depends on whether SIDE_EFFECT is evaluated before or after A (I). If evaluated before, then A (I) will raise CONSTRAINT_ERROR. If SIDE_EFFECT is evaluated after A (I) (or after the value of I is determined), then SIDE_EFFECT's value will be assigned to A (4). Regardless of the evaluation order, it is not acceptable to assign a value to a nonexistent component of A, namely A (5).

Assignment Statement Evaluation (Continued)

The RM explicitly specifies one circumstance under which evaluation order can make a program's execution erroneous, namely, a program is erroneous if the evaluation of an assignment expression changes the discriminant of a target variable:

```

type DISCRIM is (INT, FLT);

type VR (D : DISCRIM := INT) is -- a Variant Record
  record
    case D is
      when INT =>
        I : INTEGER;
      when FLT =>
        F : FLOAT;
    end case;
  end record;

R : VR; -- R has default discriminant

function F return INTEGER is
begin
  R := (D => FLT, F => 2.0);
  return 1;
end F;

R := (D => INT, I => 0);
R.I := F; -- erroneous
    
```

Since the evaluation of F changes the discriminant of R, the assignment is erroneous. Because the RM defines this situation to be erroneous, an implementation is allowed to assume that the expression does not change the discriminant of the target variable. Consequently, an implementation can evaluate R.I (checking that I exists for R's current discriminant value), evaluate F, and then assign F's value to the location that is normally occupied by R.I, even though this assignment may produce an invalid floating point value for R.F. Note that if F is evaluated first, then the evaluation of R.I will raise CONSTRAINT_ERROR, since I is not a valid selector when R.D = FLT (RM 4.1.3(8)).

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: ORDER OF EVALUATION OF PARAMETER ASSOCIATIONS

Question: In what order are parameters evaluated?

Reference: RM 6.4(6).1 - Explicit

Index Terms: *Order Dependence, Subprogram Parameters*

Rationale:

The value of actual parameter expressions may be different based on the order of evaluation chosen. The actual parameter expressions may be evaluated in any order.

Example:

In the following code fragment, side effects resulting from the order in which the parameter associations are performed may change program execution.

```
GLOBAL : INTEGER := 0;
...

function INCREMENT_GLOBAL return INTEGER is
begin
  GLOBAL := GLOBAL + 1;
  return GLOBAL;
end INCREMENT_GLOBAL;

procedure USE_GLOBAL (USE_1, USE_2 : INTEGER) is
begin
  if USE_1 = USE_2 then
    -- execution may depend upon order of evaluation of
    -- parameter associations.
    ...

  end if;
  ...

end USE_GLOBAL;
...

USE_GLOBAL (INCREMENT_GLOBAL, GLOBAL);
```

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: ORDER OF PARAMETER COPY-BACK

Question: In what order are parameters of modes **out** and **in out** copied back at the completion of a subprogram call?

Reference: RM 6.4(6).2 - Explicit

Index Terms: *Order Dependence, Subprogram Parameters*

Rationale:

The order in which parameters are copied back may affect the functional result of a program.

Example:

In the following code fragment, an exception is raised as a result of updating an **out** parameter. The value of the other **out** parameter depends upon the order in which **out** parameters are copied back.

```
procedure P (A, B : out INTEGER) is
begin
  A := -1;
  B := 5;
end P;
...

procedure Q is
  X : POSITIVE := 1;
  Y : POSITIVE := 1;
begin
  P (X, Y);
  ...

exception
  when CONSTRAINT_ERROR =>
    -- Value of Y may be either 1 or 5.
end Q;
```

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **GUARD CONDITION EVALUATION**

Question: What is the order of evaluation for guard conditions in a selective wait?

Reference: RM 9.7.1(5).1 - Explicit

Index Terms: *Order Dependence, Tasking*

Rationale:

The order of evaluation may cause different guards to be open if the evaluation of the guards has side effects.

Example:

```
...
GLOBAL_BOOLEAN : BOOLEAN := TRUE;
...
function SIDE_EFFECT return BOOLEAN is
begin
  GLOBAL_BOOLEAN := not GLOBAL_BOOLEAN;
  return GLOBAL_BOOLEAN;
end SIDE_EFFECT;
...
task body T1 is
...
begin
  select
    when GLOBAL_BOOLEAN =>
      accept A;
    or
    when SIDE_EFFECT  =>
      accept B;
  end select;
end T1;
```

Depending on the order of evaluation of the guards, the **select** statement may cause **PROGRAM_ERROR** to be raised (due to an incorrect order dependency).

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **ELABORATION ORDER OF COMPILATION UNITS**

Question: In what order are compilation units elaborated?

Reference: RM 10.5(2).1 - Explicit

Index Terms: *Elaboration, Order Dependence*

Rationale:

The elaboration of these units and of the corresponding unit bodies is performed in an order consistent with the partial ordering defined by the **with** clauses. There may be several elaboration orders which are legal and which may have different results during elaboration.

Example:

For the main program:

```
with A, B;  
procedure X is ...
```

A and B can be elaborated in any order, assuming there are no dependencies between them. The elaboration of A and B may have side effects due to common global variables that may cause different elaboration results.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: **PROGRAM_ERROR**

Question: Is **PROGRAM_ERROR** ever raised because of incorrect order dependencies?

Reference: RM 11.1(7).2 - Explicit

Index Terms: *Order Dependence, Exceptions*

Rationale:

If **PROGRAM_ERROR** is raised because of an incorrect order dependency, the user would like to know what the circumstances are so the user can handle the exception if appropriate.

Example:

Consider the following example:

```
      G : POSITIVE := 1;
      S, T : STRING (1..4);

      function F return POSITIVE is
      begin
        G := G + 1;
        return G - 1;
      end F;
      ...

      S (F..F) := T (F..F);
```

The calls to the function F will result in a different slice of the string T being assigned into the string S, depending on the order of evaluation for the operands of the assignment statement. An implementation could choose to raise **PROGRAM_ERROR** in this situation.

Catalogue of Ada Runtime Implementation Dependencies - Order Dependence

TOPIC: ELABORATION OF GENERIC INSTANTIATIONS

Question: In what order are explicit generic actual parameters evaluated?

Reference: RM 12.3(17).1 - Explicit

Index Terms: *Order Dependence, Elaboration*

Rationale:

For each elaboration of a generic instantiation, expressions (and names, see AI-00365) that are supplied as explicit generic actual parameters or as constituents of variable names or entry names may be evaluated in some order not defined by the language. For expressions with side effects, the order of evaluation may affect the results of the program.

Example:

Consider the following example:

```
generic
  A, B : in out INTEGER;
procedure G_TEST is
begin
  ...
end G_TEST;

type TEST_ARRAY is array (INTEGER range < >) of INTEGER;

G : INTEGER := 0;
H : TEST_ARRAY (0..3);

function F return INTEGER is
begin
  G := G + 1;
  return G - 1;
end F;
...

procedure G_TEST_1 is new G_TEST (A => H (F), B => H (F));
```

Depending on the order of evaluation, A is associated with the zeroth element of the array and B is associated with the first element of the array or vice versa.

Chapter 9

Erroneous Programs

TOPIC: EFFECT OF ERRONEOUS EXECUTION

Question: What is the effect of erroneous execution?

Reference: RM 1.6(7).1 - Explicit

Index Terms: *Erroneous Execution*

Rationale:

The RM states that "the effect of erroneous execution is unpredictable". However, each implementation will deal with specific instances of erroneous code in (generally) predictable ways. Consequently, the user may wish to know the effect of erroneous program execution. This question applies for all specific instances indicated as questions in other chapters.

Example:

```
...
A : INTEGER; -- With the assumption the compiler will
              -- initialize all integer variables to the
...          -- integer value 0
```

If correct execution of the program depends on the assumed value of A, the program is erroneous. Thus, given a procedure with the following specification:

```
procedure ABC (IN_PARAM : in out POSITIVE);
...
-- The following sequence is erroneous:
A : POSITIVE; -- Value is undefined
...
ABC (IN_PARAM = > A); -- Value of A is undefined
```

Catalogue of Ada Runtime Implementation Dependencies - Erroneous Programs

TOPIC: EVALUATION OF SCALAR VARIABLES

Question: What is the value of a scalar variable that has not been initialized?

Reference: RM 3.2.1(17).1, 3.2.1(18).1 - Explicit

Index Terms: *Undefined Values, Erroneous Execution*

Rationale:

The execution of a program is erroneous if it attempts to reference a scalar variable with an undefined value, or attempts to apply a predefined operator to a variable with a scalar subcomponent having an undefined value.

Example:

Given a procedure with the following specification:

```
procedure ABC (IN_PARAM : in out POSITIVE);  
...
```

-- The following sequence is erroneous:

```
A : POSITIVE; -- Value is undefined  
...
```

```
ABC (IN_PARAM = > A); -- Value of A is undefined
```

Catalogue of Ada Runtime Implementation Dependencies - Erroneous Programs

TOPIC: **PROGRAM_ERROR**

Question: Is PROGRAM_ERROR ever raised because of an erroneous program?

Reference: RM 11.1(7).1 - Explicit

Index Terms: *Erroneous Execution, Exceptions, Runtime Checking*

Rationale:

If PROGRAM_ERROR is raised when an erroneous action occurs, the user would like to know what the circumstances are, so the user can handle the exception if appropriate.

Example:

Consider the following example:

```
type DISCRIM is (INT, FLT);
type VR (D : DISCRIM := INT) is
  record
    case D is
      when INT =>
        I : INTEGER;
      when FLT =>
        F : FLOAT;
    end case;
  end record;

R : VR;
...

function F return INTEGER is
begin
  R := (D => FLT, F => 2.0);
  return 1;
end F;
...

R := (D => INT, I => 0);
R.I := F;        -- erroneous
```

Since the evaluation of F changes the discriminant of R, the assignment is erroneous (RM 5.2(4)). Since this assignment is known to be erroneous, an implementation could choose to raise PROGRAM_ERROR.

Catalogue of Ada Runtime Implementation Dependencies - Erroneous Programs

TOPIC: **ACCESSING UNCHECKED DEALLOCATED OBJECTS**

Question: What is the effect of accessing an unchecked deallocated object through a redundant designator?

Reference: RM 13.10.1(6).1 - Implicit

Index Terms: *Erroneous Execution, Access Type*

Rationale:

The RM states that an deallocated object which is accessed through a redundant designator is erroneous. While the effect of such an access is not defined by the language, an implementation will choose some action to be performed to handle this problem at runtime.

Example:

An implementation may choose to provide an elaborate runtime check to ensure there are no other references when the **FREE** procedure is invoked. That is, when X and Y designate the same object and **FREE (X)** is executed, the runtime may supply a special value to Y, which indicates the object is no longer available. If Y is not changed, the next access through Y would be capable of raising an exception.

Chapter 10

Exception Issues

TOPIC: EXCEPTION REPRESENTATION

Question: How are exceptions represented?

Reference: RM 11.1(1).1 - Implicit

Index Terms: *Exceptions, Optimization (Time and Space)*

Rationale:

An exception must be unambiguously represented in an executing Ada program. How the mapping from the exception name to an underlying representation is done may have an impact on the speed and size of the generated code.

Example:

Some implementations represent an exception as a pointer to a string containing the actual exception name. This is useful for debugging and error reporting. However, for an embedded system where no textual I/O is available, it may be wasteful to keep the string representation of every exception name in memory.

Catalogue of Ada Runtime Implementation Dependencies - Exception Issues

TOPIC: **IMPLEMENTATION-DEFINED EXCEPTIONS**

Question: What additional exceptions are provided?

Reference: RM 11.1(4).1 - Implicit

Index Terms: *Exceptions*

Rationale:

Users may wish to handle these exceptions. As such, it is necessary to know what these exceptions are and under what circumstances they can be raised.

Example:

One Ada implementation defines an exception `NON_ADA_ERROR` in package `SYSTEM`. This exception is raised when non-Ada code is imported into an Ada program, and that code then signals a condition. The user can then explicitly handle this exception in an exception handler in the Ada code.

Catalogue of Ada Runtime Implementation Dependencies - Exception Issues

TOPIC: **EXCEPTION HANDLER OVERHEAD**

Question: What is the overhead if a frame has an exception handler associated with it?

Reference: RM 11.2(3).1 - Implicit

Index Terms: *Exceptions, Optimization (Time)*

Rationale:

Since exceptions are used to indicate "exceptional situations", exception handlers should not be executed during normal program execution. It may therefore be desirable to have minimal overhead associated with entering and exiting a frame that has an exception handler.

Example:

Consider the following example (Reference RM, 11.4.1(11)):

```
function FACTORIAL (N : POSITIVE) return FLOAT is
begin
  if N = 1 then
    return 1.0;
  else
    return FLOAT (N) * FACTORIAL (N-1);
  end if;
exception
  when NUMERIC_ERROR => return FLOAT'SAFE_LARGE;
  -- SEE AI-00387
end FACTORIAL;
```

If the overhead for entering a frame with an exception handler is significant, the execution time for this function might be higher than if the function had been written to explicitly check the operands of the multiply before performing the multiply. If the overhead were low, the above example would execute faster than if it had been coded with explicit checks.

Catalogue of Ada Runtime Implementation Dependencies - Exception Issues

TOPIC: NON-ADA EXCEPTIONS

Question: Are "exceptions" which are raised in non-Ada subprograms handled by Ada exception handlers?

Reference: RM 11.2(6).1 - Implicit

Index Terms: *Exceptions, Foreign Languages*

Rationale:

When a non-Ada subprogram is called, the user must be aware of whether or not it can raise any exceptions. If it can, the user must know which exceptions it can raise, how they are raised and if they are propagated to the Ada unit that called the non-Ada code.

Example:

Assume a non-Ada function is called from an Ada subprogram. During the execution of the function a divide by zero is performed. If the non-Ada language does not support a mechanism for handling such an exception, the exception might be propagated as `NUMERIC_ERROR` (SEE AI-00387) to the calling Ada subprogram.

TOPIC: MAIN PROGRAM TERMINATION

Question: What happens when the execution of the main program is abandoned after an unhandled exception?

Reference: RM 11.4.1(5).1, 11.4.1(20).1 - Explicit

Index Terms: *Program Termination, Exceptions*

Rationale:

A user may desire to intercept such exceptions "outside of" the Ada program.

Example:

On many host systems a message is written out to a standard output device. Or perhaps on an embedded system, the failure of an Ada program for a specific exception could trigger the execution of an error recovery program.

Catalogue of Ada Runtime Implementation Dependencies - Exception Issues

TOPIC: CODE MOTION

Question: How do optimizing code movement and exceptions interact to affect the results of code sequences?

Reference: RM 11.6(11).1 - Implicit

Index Terms: *Optimization (Time), Exceptions*

Rationale:

Certain code movements could cause an exception to occur earlier than its textual position would indicate. If this situation can arise, the user may want to enclose statements within a block to get control over where the exception is raised.

Example:

Consider the following example (cf. Ada Implementers Guide, pp. 11-14). Since the order of operand evaluation is not defined by the language, given:

```
B : array (1..30) of INTEGER;  
A, D : INTEGER;  
  
begin  
  D := 30;  
  if A = D / (A - 3) or A = B (50) then ...  
  
exception  
  ...
```

there is no guarantee that `NUMERIC_ERROR` (SEE AI-00387) will be raised instead of `CONSTRAINT_ERROR` if `A = 3`. Either operand of the `or` statement can be evaluated first. Moreover, if `CONSTRAINT_ERROR` is raised, there is no guarantee that `D = 30` in a handler for this exception, since `B (50)` can be evaluated before the assignment to `D` is performed.

Catalogue of Ada Runtime Implementation Dependencies - Exception Issues

TOPIC: PRAGMA SUPPRESS

Question: For each check that can be suppressed, what is the effect (or possible effect) of an error arising with that check being suppressed?

Reference: RM 11.7(18).1 - Explicit

Index Terms: *Optimization (Time), Exceptions, Runtime Checking*

Rationale:

There may be situations where pragma SUPPRESS has been used to prevent a check, and an error condition arises. In these cases, it would be helpful to know what the possible effects are.

Example:

If OVERFLOW_CHECK is suppressed, some implementations of floating point operations may always flag the floating point number as "not a number". Other implementations may simply generate a truncated result.

TOPIC: IMPLEMENTATION-DEFINED EXCEPTIONS

Question: What implementation-defined exceptions are available?

Reference: RM 14.1(11).1 - Explicit

Index Terms: *Exceptions, Input-Output*

Rationale:

The availability and use of implementation-defined exceptions may potentially affect the performance and transportability of a program unit.

Example:

A program unit that depends upon an implementation-defined exception for input- output processing may not be transportable when combined with a program unit that is executed under an implementation that does not support the exception.

Chapter 11

Input-Output Issues

TOPIC: FORM PARAMETER

Question: What system-dependent characteristics defined in the FORM parameter can affect input-output operations?

Reference: RM 14.1(1).1 - Explicit

Index Terms: *Input-Output*

Rationale:

The effects of system-dependent characteristics defined in the FORM parameter may potentially affect input-output operations. Dependence upon these effects may potentially affect the transportability of a program unit.

AI-00278 also clarifies that: "A FORM parameter can be used to obtain the effect of appending to a text or sequential file."

Example:

A program may depend upon the FORM parameter to control shared access to an external file. This program may not be reusable/transportable under implementations that do not support the explicit sharing/nonsharing of files.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: EXTERNAL FILE STATUS

Question: What is the disposition of the external file associated with a non-closed file following completion of the main program?

Reference: RM 14.1(7).1 - Explicit

Index Terms: *Input-Output*

Rationale:

The disposition of the external file associated with a file that is not closed prior to completion of the main program in which it was opened may potentially affect the execution of subsequent programs. Dependence upon a specific disposition can potentially affect the transportability of a group of programs.

Example:

A program that depends upon the implementation to close all opened files upon program completion may result in external files being unavailable to subsequent programs under implementations that do not close opened files and leave the associated external files as unavailable, i.e., currently in use.

TOPIC: INPUT-OUTPUT OF ACCESS TYPE OBJECTS

Question: What is the effect of input-output for access type objects?

Reference: RM 14.1(7).2 - Explicit

Index Terms: *Input-Output, Optimization (Time)*

Rationale:

The facility to input and output access type objects may potentially affect the performance, reusability, and transportability of a program unit.

Example:

A program that depends upon a facility to output and then subsequently input access type objects may not be transportable to an implementation that raises `USE_ERROR` exception when an access type object is output.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: CONCURRENT SHARING-OF EXTERNAL FILES

Question: What are the effects of associating more than one file with a single external file?

Reference: RM 14.1(13).1 - Explicit

Index Terms: *Input-Output, Tasking*

Rationale:

The effects of sharing an external file among more than one file can affect input-output operations. Dependence upon these effects may potentially affect the reusability, and transportability of a program unit.

Example:

A program that depends upon the facility to share an external file among concurrently executing tasks may not be transportable to an implementation where an external file may only be associated with a single file. In such implementations, once an association is made, subsequent attempts to associate the external file with another file may raise exceptions.

See also 14.3.5(3).1

TOPIC: NULL FORM ARGUMENT STRING

Question: What is the effect of a null FORM argument string?

Reference: RM 14.2.1(3).3 - Implicit

Index Terms: *Input-Output*

Rationale:

The use of a null FORM argument string may potentially affect the reusability and transportability of a program unit when a file is created or opened.

Example:

A program that depends upon the implementation to provide default options for external files may not be transportable or reusable under implementations that require a non-null FORM argument string when creating or opening a file. In these implementations the exception `USE_ERROR` is raised.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: FILE I/O NOT SUPPORTED, EXCEPTIONS FOR

Question: What exception is raised if file IO is not supported and a file is opened?

Reference: RM 14.2.1(3).4 - Implicit

Index Terms: *Input-Output, Exceptions*

Rationale:

AI-00332 indicates that:

"CREATE and OPEN can raise USE_ERROR or NAME_ERROR if file creation or opening is not allowed for any file."

"In discussing this issue, it was noted that when no files can be created or opened, every attempt to create a temporary file will raise USE_ERROR; NAME_ERROR cannot be raised because no name is specified."

This points out two dependencies: does the implementation support file IO, and what exception is raised if a file operation is attempted and if IO is not supported?

Example:

Exception handlers for systems that potentially could operate on runtime environments which cannot support file IO must compensate for either possibility:

```
exception
  when NAME_ERROR | USE_ERROR =>    -- maybe file IO is not supported!
```

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: CLOSING TEMPORARY FILES

Question: What is the status of a temporary file after it has been closed?

Reference: RM 14.2.1(9).1 - Implicit

Index Terms: *Input-Output, External Files*

Rationale:

The status of the external file associated with a temporary file that has been closed may potentially affect the performance, reusability and transportability of a program unit.

AI-00046 states:

- "1) An implementation is allowed to delete a temporary file immediately after closing it.
- 2) The NAME function is allowed to raise `USE_ERROR` if its argument is associated with an external file that has no name, in particular, a temporary file."

Example:

A program that depends upon the facility to access the external file associated with a temporary file, after it has been closed, may not be reusable or transportable to an implementation that does not support the facility. The following code fragment illustrates the dependency:

```
CREATE(FILE,OUT_FILE,"");
FILE_NAME := NAME(FILE); -- USE_ERROR?
-- Assumes that NAME will return the name of
-- the external file
...
CLOSE(FILE);
OPEN(FILE,IN_FILE,FILE_NAME);
-- NAME_ERROR may be raised if the external
-- file ceased to exist as a result of closing
-- the temporary file.
```

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: CLOSING SEQUENTIAL FILES

Question: Is a file terminator placed at the current position of a sequential file when it is closed?

Reference: RM 14.2.1(9).2 - Implicit

Index Terms: *Input-Output, External Files*

Rationale:

NOTE: THIS IMPLEMENTATION DEPENDENCY HAS BEEN MADE OBSOLETE BY AI-00357. It is included here for completeness.

Although it is clearly stated for Text_IO files, the RM fails to indicate if a sequential file should have a file terminator placed at the current position when the file is closed (or reset) when opened for output. AI-00357 states:

"If a sequential input-output file having mode OUT_FILE is closed or reset, the most recently written element since the last open or reset is the last element that can be read from the file. If no elements have been written, the closed or reset file is empty. (As a consequence, opening a sequential input-output file with mode OUT_FILE or resetting a sequential input-output file to mode OUT_FILE has the effect of deleting the previous contents of the file.)"

Example:

Consider the following sequence of actions for a sequential file:

1. Create file for output.
2. Write 10 elements into the file.
3. Close the file.
4. Open the file for output.
5. Write 5 elements into the file.
6. Reset the file for input.
7. Read elements until the end of the file.

Step 7 could read possibly read 5 new values, plus 5 old values.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **DELETION OF SHARED EXTERNAL FILES**

Question: What is the effect of attempting to delete a file that is associated with a shared external file?

Reference: RM 14.2.1(13).1 - Implicit

Index Terms: *Input-Output*

Rationale:

The effect of attempting to delete a file that is associated with an external file that is currently associated with another file may potentially affect the reusability, and transportability of a program unit.

Example:

A program that depends upon a specific exception to be raised as a result of deleting a shared external file may not be transportable to an implementation that does not raise the same exception. The following code fragment illustrates the dependency:

```
FILE_NAME := EXTERNAL_NAME;
OPEN (FILE_1, IN_FILE, FILE_NAME);
OPEN (FILE_2, IN_FILE, FILE_NAME);
-- Exception may be raised if sharing an external file
-- is not supported.
...

DELETE (FILE_1);
-- Exception may be raised if FILE_2 has not been closed.
...

READ (FILE_2, ITEM);
-- Exception may be raised if DELETE was performed.
```

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **NAME OF AN EXTERNAL FILE ASSOCIATED WITH A TEMPORARY FILE**

Question: What is the effect of calling the function NAME with a temporary file?

Reference: RM 14.2.1(21).1 - Implicit

Index Terms: *Input-Output*

Rationale:

Depending on the effect of attempting to obtain the name of the external file associated with a temporary file may potentially affect the reusability, and transportability of a program unit.

Example:

A program that depends upon the function NAME to obtain the name of the external file associated with a temporary file may not be reusable or transportable to an implementation that does not support the facility. Refer to CRID 14.2.1(9).1.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: UN-INTERPRETABLE ELEMENTS

Question: What is the effect of reading an element that cannot be interpreted as a value of the expected type?

Reference: RM 14.2.2(4).1, 14.2.4(4).1 - Explicit

Index Terms: *Input-Output, Exceptions*

Rationale:

Depending on the effect of reading an element that cannot be interpreted as a value of the expected type may potentially affect the reusability and transportability of a program unit.

Example:

A program unit that depends upon the exception `DATA_ERROR` to be raised when reading an element that cannot be interpreted as a value of the expected type may not be reusable when combined with a program unit that is executed under an implementation that does not raise the exception. The following code fragment illustrates the dependency:

```
declare
  function READ_DATA return ELEMENT_TYPE is
  ...

begin
  ...

  READ (FILE, ITEM);
  -- DATA_ERROR is assumed to be raised
  -- for un-interpretable value for ITEM.
  return ITEM;
end READ_DATA;

begin
  ...

  ITEM := READ_DATA;
  -- Data validity dependent on DATA_ERROR.
exception
  when DATA_ERROR =>
    -- Invalid data dependency.
end;
```

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **TERMINATORS**

Question: What are the effects of end of a line, page, and file terminators on the input and output of control characters?

Reference: RM 14.3(7).1 - Explicit

Index Terms: *Input-Output*

Rationale:

The use of the effects of the representation of line, page, and file terminators may potentially affect the reusability and transportability of a program unit.

Example:

A program unit that depends upon a specific representation for line, page, and file terminators may not be reusable when combined with a program unit that is executed under an implementation that does not support this representation.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **WAITING FOR PAGE TERMINATORS**

Question: What is the effect of reading a line terminator from an interactive device using GET_LINE and SKIP_LINE?

Reference: RM 14.3.4(8).1, 14.3.5(13).1 - Implicit

Index Terms: *Input-Output*

Rationale:

The use of the effect of reading a line terminator from an interactive device may potentially affect the reusability and transportability of a program unit. An implementation may choose either to wait for a subsequent page terminator or to continue when the external file represents an interactive device and does not contain page terminators.

Example:

A program unit that depends upon GET_LINE and SKIP_LINE returning as soon as a line terminator is read for an external file that represents an interactive device may not be transportable under an implementation that always checks for the presence of a page terminator following the line terminator. In such an implementation, additional input must be read before GET_LINE and SKIP_LINE can return.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **SHARING EXTERNAL FILES**

Question: What interactions are possible when the same external file is shared by two or more file objects?

Reference: RM 14.3.5(3).1 - Implicit

Index Terms: *Input-Output, Shared Files*

Rationale:

AI-00320 states:

"If several file objects are associated with the same external file, some effects are implementation dependent."

Example:

Since the effect of sharing an external file is not fully defined by the Standard, if an external file is associated with two **sequential** file objects having mode **IN_FILE**, a read operation using one of the file objects could cause **END_OF_FILE** to become true for the other object. Similarly, the data read for one of the file objects could be affected by how many read operations have been applied to the other object. Some effects of sharing external files, however, are specified explicitly. In particular, 14.2(4) says, "The current index of a **direct** file is a property of a file object, not of an external file." This wording is intended to mean that if the same external file is associated with two **direct** file objects, each file object maintains its own current index for purposes of reading and writing elements of the external file. A read operation applied to one file object will not affect the current index of the other file object.

See also 14.1(13).1

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **BUFFERING FILE INPUT AND OUTPUT**

Question: What is the effect of file buffering on text input and output operations?

Reference: RM 14.3.6(1).1, 14.3.6(7).1 - Implicit

Index Terms: *Input-Output*

Rationale:

The use of the effect of file buffering may potentially affect the reusability and transportability of a program unit. Implementations may buffer text so that the result of an input or output operation may be delayed.

Example:

A program unit that depends upon the immediate result of an output operation may not be reusable under an implementation where the operation is delayed pending the next input or output operation. The following code fragment illustrates the dependency:

```
...  
  
PUT (CONTROL_CONSOLE, "Shutdown Nuclear Reactor NOW! ");  
-- Dependency on non-buffered input so that  
-- text is displayed in a timely manner.  
  
delay SHUTDOWN_PERIOD;  
if SHUTDOWN_SUCCESSFUL then  
  PUT_LINE (CONTROL_CONSOLE, " -- Shutdown completed");  
else  
  raise INSURANCE_LIABILITY;  
end if;
```

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **END OF LINE AND END OF STRING**

Question: What is the effect of reading a line when the end of string coincides with the end of line?

Reference: RM 14.3.6(13).1 - Implicit

Index Terms: *Input-Output*

Rationale:

THIS IMPLEMENTATION DEPENDENCY HAS BEEN ELIMINATED DUE TO AI-00050, which states:

"GET_LINE reads characters until either the end of the string is met or until END_OF_LINE is true. If the end of the string has been met, SKIP_LINE is not called even if END_OF_LINE is true. In particular, no characters are read if the string is null."

The following text is left in the catalogue for completeness:

The use of the effect of reading a line where the end of string coincides with the end of line may potentially affect the reusability and transportability of a program unit. An implementation may or may not call SKIP_LINE under these conditions. (PRIOR TO AI-00050)

Example:

A program unit that depends upon the effect of GET_LINE calling SKIP_LINE when the end of line coincides with the end of string may not be transportable to an implementation that does not call SKIP_LINE under the same conditions. The following code fragment illustrates the dependency:

```
TEXT_IO.SET_LINE_LENGTH (TEXT_FILE, 0);
...
for NEXT_STRING in TEXT_ARRAY'RANGE loop
  TEXT_IO.PUT_LINE (TEXT_FILE, TEXT_ARRAY (NEXT_STRING),
    LAST);
end loop;
...
for NEXT_STRING in TEXT_ARRAY'RANGE loop
  TEXT_IO.GET_LINE (TEXT_FILE, TEXT_ARRAY (NEXT_STRING));
  -- Dependency on GET_LINE calling SKIP_LINE
  -- if all of the above output strings are
  -- to be input.
end loop;
```

An implementation that elects not to call SKIP_LINE when the end of string is reached causes the latter code fragment to input strings only on every other GET_LINE, because a GET_LINE is used to process each line terminator.

TOPIC: REPRESENTATION OF NON-GRAPHIC CHARACTERS

Question: What are the effects of IO of non-graphic characters?

Reference: RM 14.3.9(6).1 thru (9) - Implicit

Index Terms: *Input-Output, Exceptions*

Rationale:

NOTE: THIS IMPLEMENTATION DEPENDENCY IS EFFECTED BY THE NON-BINDING INTERPRETATION STATED IN AI-00239:

"If `ENUM_IO` is an instantiation of `ENUMERATION_IO` for a character type that contains a non-graphic character, e.g.,

```
package ENUM_IO is new ENUMERATION_IO (CHARACTER);
```

then for each non-graphic character (such as `ASCII.NUL`), `ENUM_IO.PUT` should output the corresponding sequence of characters used in the type definition (e.g., `PUT(ASCII.NUL)` should output the string "NUL" if `SET` has the value `UPPER_CASE` and `WIDTH` is less than 4). Furthermore, `ENUM_IO.GET` should be able to read the sequence of characters output by `ENUM_IO.PUT` for a non-graphic character, returning in its `ITEM` parameter the corresponding enumeration value.

Similarly, the image of a non-graphic character (i.e., the result returned for the attribute designator `IMAGE`) should be the sequence of characters used in the type definition of `CHARACTER` (e.g., `CHARACTER'IMAGE(ASCII.NUL) = "NUL"`), and `'VALUE` should accept such a string as representing the corresponding enumeration value.

An implementation conforms to the Standard in this respect if the result produced by `'IMAGE` for a non-graphic character is accepted by `'VALUE`, and if the result (if any) produced by `PUT` can be read by `GET`; `GET` is also allowed to raise `DATA_ERROR` when attempting to read any string produced by `PUT` for a non-graphic character.

This interpretation is non-binding, i.e., implementers are encouraged to conform to it but are not required to do so by the validation tests. A future version of the Standard may incorporate this interpretation."

The use of the effects of reading a non-graphic character may potentially affect the reusability and transportability of a program unit. The representation of non-graphic characters when `ENUMERATION_IO` is instantiated for the type `CHARACTER` may vary. Furthermore, a non-graphic character that has been output using `PUT` may raise the exception `DATA_ERROR` when input using `GET`.

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

Representation of non-graphic characters (continued)

Example:

A program unit that depends upon a specific representation of a non-graphic character may not be reusable when combined with a program unit that is executed under an implementation that does not support the same representation. The following code fragment illustrates the dependency:

```
with TEXT_IO;
package ENUM_IO is new TEXT_IO.ENUMERATION_IO (CHARACTER);

function IS_NULL (FILE : TEXT_IO.FILE_TYPE) return BOOLEAN is
  ITEM : CHARACTER;
begin
  ENUM_IO.GET (FILE, ITEM);
  -- DATA_ERROR may occur in an implementation where
  -- the function is reused does not support the input
  -- of non-graphic characters.

  return ITEM = ASCII.NUL;
end IS_NULL;
...

TEXT_IO.CREATE (ENUM_FILE, OUT_FILE, "NULL_FILE");
ENUM_IO.PUT (ENUM_FILE, ASCII.NUL);
-- Output of the null character does not guarantee
-- that it can be successfully read by a different
-- implementation.

TEXT_IO.CLOSE (ENUM_FILE);
TEXT_IO.OPEN (ENUM_FILE, IN_FILE, "NULL_FILE");
NULL_FILE := IS_NULL (ENUM_FILE);
...
```

Catalogue of Ada Runtime Implementation Dependencies - Input-Output

TOPIC: **IMAGE OF NON-GRAPHIC CHARACTERS**

Question: What is the IMAGE of a non-graphic character?

Reference: RM Annex A(18).1 - Implicit

Index Terms: *Input-Output, Attributes*

Rationale:

This implementation dependency is effected by the non-binding interpretation stated in AI-00239. For the text of this AI, refer to 14.3.9(6).1 in the chapter on Input-Output Issues.

The RM does not define the IMAGE of non-graphic characters; this is left to the implementation.

Example:

The RM defines the list of graphic characters in Section 2.1 on the Character Set. All other characters are non-graphic characters.

Note:

Several of the other predefined attributes are implementation dependent in an absolute sense; but they are not in a relative sense. For example, P'POSITION may be positive or negative depending on an implementation; but adding its value to the starting address of the enclosing record object will always yield the address of the component P regardless of the sign of the POSITION value. Should these be added to the list of implementation dependencies? The others include: ADDRESS, FIRST_BIT, LAST_BIT, MACHINE_EMAX, MACHINE_EMIN, MACHINE_MANTISSA, MACHINE_RADIX, STORAGE_SIZE, and STORAGE_SIZEF.

See also 14.3.9(6).1

Chapter 12

Other Issues

TOPIC: ALLOWABLE CHARACTERS IN COMMENTS

Question: Are characters other than the 7-bit ISO set allowed in comments?

Reference: RM 2.1(1).1 and 2.7(1) - Implicit

Index Terms: *Comments, Character Set*

Rationale:

AI-00339 indicates:

"An implementation is allowed (but not required) to accept an extended character set (i.e., graphic characters whose codes do not belong to the ISO seven-bit coded character set (ISO standard 646)) as long as the additional characters appear only in comments."

Example:

The 8-bit character set makes it possible to write comments in other than the English language.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: PRAGMAS, EXPRESSIONS WITHIN

Question: What is the meaning of identifiers and operations within implementation defined pragmas?

Reference: RM 2.8(7).1 - Implicit

Index Terms: PRAGMA

Rationale:

AI-00010 indicates: "If a name given in a pragma argument is both visible at the place of the pragma and an identifier specific to the pragma, the meaning of the name depends on the pragma.

An operation given in a pragma argument must be either visible at the place of the pragma or have a meaning that is specific to the pragma. If the operation is both visible at the place of the pragma and has a meaning that is specific to the pragma, the chosen meaning depends on the pragma.

For an implementation-dependent pragma, the meaning of an expression given as an argument depends on the pragma."

Example:

`CUR_LINE : INTEGER := 4; pragma ALIGN_PAGE (CUR_LINE + 3);`

An implementation can specify that the meaning of CUR_LINE is implementation-defined if CUR_LINE is not a visible name; the implementation could further specify that if CUR_LINE is a visible name, the visible meaning is used in preference to the implementation-defined meaning, or vice versa.

Further, is the predefined "+" operation used or is a visible "+" operator used, or even an implementation-defined "+" operation?

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: IMPLEMENTATION-DEFINED ATTRIBUTES

Question: What attributes, in addition to those in Annex A, does the implementation support?

Reference: RM 4.1.4(4).1 - Explicit

Index Terms: *Attributes*

Rationale:

New attributes provided by an implementation may provide values that are unique to the runtime environment of that implementation.

Example:

An implementation may provide several additional addressing attributes that extract the base register and displacement components of an address, namely, A'BASE_REGISTER and A'DISPLACEMENT. These attributes would then be applied to the value produced by the ADDRESS attribute, as these examples show:

...X'ADDRESS'BASE_REGISTER...-- produces the base register number
...X'ADDRESS'DISPLACEMENT... -- produces the displacement value

Such attributes may not be found in other implementations.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: VALUE OF SCALAR OUT PARAMETERS

Question: For subprograms that do not update scalar out parameters, what is the value upon return from the subprogram?

Reference: RM 6.2(5).1 - Explicit

Index Terms: *Subprogram Parameters, Undefined Values*

Rationale:

The RM states that the value of a scalar out parameter that is not updated by the called subprogram is undefined upon return. There are several different actions a compiler may take in this situation, each resulting in a different runtime behavior of the program. Some of the possible actions include:

- a) copy back whatever bit pattern is in the storage location associated with the parameter (possibly causing a `CONSTRAINT_ERROR` exception),
- b) omission of the copy-back operation,
- c) raise the `PROGRAM_ERROR` exception at runtime due to referencing an uninitialized formal parameter.

In addition, a compile-time diagnostic message could be generated. It may not be possible to detect all instances of unreferenced out parameters prior to program execution.

Example:

In the following code fragment, the actual parameters associated with A and B may not be referenced; however, there is no guarantee that their respective values will remain the same.

```
procedure P (A : out POSITIVE; B : out POSITIVE) is
  LOCAL : POSITIVE;
begin
  ...
  if SOME_GLOBAL_BOOLEAN then
    -- Only reference to formal parameters.
    A := LOCAL;
    B := LOCAL + 1;
  end if;
end P;

...
X := Y;
P (X, Y);
if X = Y then
  -- Result of conditional expression is unpredictable.
end if;
```

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: PACKAGE STANDARD

Question: Is a single package STANDARD implied for the compilation of a distributed Ada program?

Reference: RM 8.6(1).1 - Implicit

Index Terms: *Package STANDARD*

Rationale:

The support of heterogeneous distributed processors may depend upon the availability of multiple versions of the package STANDARD.

Example:

A compilation system that targets both a MIL-STD-1750A and an INTEL 80386, communicating over a common bus, should have two very different package STANDARDS. This is because of the difference in word sizes, addresses, etc.

TOPIC: MAIN PROGRAM INITIATION

Question: What conventions are established by the environment task prior to execution of the main program?

Reference: RM 10.1(8).1 - Explicit

Index Terms: *Program Initiation*

Rationale:

The user may require knowledge of what state was established for the environment, and what resources are available to the user.

Example:

The environment may initiate program execution with interrupts enabled or disabled.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: MAIN PROGRAM PARAMETERS AND RESULTS

Question: What restrictions are placed on the parameters and results of a main program?

Reference: RM 10.1(8).2 - Explicit

Index Terms: *Program Initiation, Subprogram Parameters*

Rationale:

A parameterless procedure is capable of being a main program, but other types of subprograms may be main programs also. The facilities provided for a main program to receive information from its external environment and to return information back affect the design of systems.

Example:

The external environment may have a convention where a set of state variables is presented to each main program to be updated. These state variables are supplied as in out parameters of all main programs. Alternatively, an external environment may have a convention that all main programs must return a function value as a status code.

TOPIC: ELABORATION PRIOR TO PROGRAM EXECUTION

Question: To what extent is elaboration accomplished prior to execution?

Reference: RM 10.5(2).2 - Implicit

Index Terms: *Static Allocation, Program Initiation, Elaboration*

Rationale:

Code explicitly generated to accomplish elaboration generally constitutes overhead that might be avoided through the creation of pre-elaborated values for program variables and constants.

Example:

```
package PACK1 is
  C : constant INTEGER := 6;
end PACK1;
```

In this case there might be no overhead involved in the elaboration of the object C.

TOPIC: ASSIGNMENT STATEMENT CONSTRAINT CHECKING

Question: In an assignment statement, is the check that is made to determine if the value of an expression belongs to the subtype of the target variable ever made prior to the complete evaluation of the expression?

Reference: RM 11.6(4).1 - Explicit

Index Terms: *Runtime Checks, Exceptions, Order Dependence*

Rationale:

Although the RM states that the value of the expression is checked against the subtype of the variable after the expression is evaluated, an optimizer can perform the subtype check earlier. RM 11.6(4) allows predefined operations in assignment statements to be evaluated as soon as possible, and the act of checking a discriminant constraint or array length involves the use of predefined equality operations.

Due to the explicit wording in section 5.2(3) some confusion may exist about what behavior can exist. If a user is not familiar with Chapter 11, they may not be aware of possible optimizations which supersede 5.2(3). Transporting software may also cause problems to appear that are difficult to isolate in the absence of information on what optimizations are done with constraint checking.

Example:

```
type ARR is array (INTEGER range < >) of INTEGER;
type REC (D : INTEGER) is
  record
    X : INTEGER;
  end record;

ARR_V : ARR (1..3);
REC_V : REC (3);
GLOBAL : INTEGER := 1;

function F return INTEGER is
begin
  GLOBAL := GLOBAL + 1;
  return GLOBAL;
end F;

ARR_V := (1..4 => F);
REC_V := (D => 4, X => F);
```

F need not be invoked in either of these assignments since in the first assignment, the length of the array value can be determined to exceed the length of ARR_V prior to evaluating F, and similarly, in the second assignment, the discriminant of the aggregate can be determined to be unequal to the discriminant of REC_V before F is invoked. In both cases, CONSTRAINT_ERROR can be raised before F is evaluated. Consequently, GLOBAL can have the value 1 when an exception handler is entered.

TOPIC: SHARED CODE FOR GENERICS

Question: Are generics implemented using shared code?

Reference: RM 12.3(5).1 - Implicit

Index Terms: *Optimization (Space)*

Rationale:

If each implementation of a generic unit requires its own separate instance of the generated code, a program may use more space than if the same code is reused (to the extent applicable) for each instantiation. Conversely, there may be an execution time penalty for certain types of code sharing. The criteria used for code sharing and the ways to affect the degree of code sharing should be known to the user.

Example:

If a user writes a generic simple sort and then instantiates it for five different integer types, the executable image size and execution performance are likely to be affected by the choices made by the compiler as to whether it generates five integer sorts (and potentially many others for different types). In some situations where the generic is very small and time is critical, the user may prefer to have multiple copies of the code. In other instances, particularly if the generic is very large, the user may prefer the time penalty if a single copy of the generic instantiation is created.

TOPIC: EXPRESSIONS IN REPRESENTATION CLAUSES

Question: What is the interpretation of expressions that appear in representation clauses?

Reference: RM 13.1(10).1 - Explicit

Index Terms: *Representation Clauses*

Rationale:

The interpretation of the expressions that appear in representation clauses may affect the portability of the user's code.

Example:

An implementation may have a convention that distinguishes the value of an expression as a virtual address or as a physical address.

TOPIC: ACCEPTANCE OF REPRESENTATION CLAUSES

Question: What representation clauses are accepted by an implementation?

Reference: RM 13.1(10).2 - Explicit

Index Terms: *Representation Clauses*

Rationale:

If a program includes representation clauses that are not accepted by a particular implementation, this will cause that program to be illegal on that implementation. This would affect the portability of that program.

Example:

An implementation may not accept any representation clauses.

TOPIC: PACKING ALGORITHM FOR THE PRAGMA PACK

Question: What is the algorithm used for minimizing the space between components of a type specified in the **pragma PACK**?

Reference: RM 13.1(12).1 - Implicit

Index Terms: *Representation Clauses, Optimization (Space)*

Rationale:

The RM defines packing to mean that the gaps between the storage areas allocated to consecutive components of record or array types should be minimized. Yet from one implementation to another even on the same underlying system the size of the gaps may be different. The algorithm to implement **PACK** determines its utility on a large number of applications.

Example:

Consider a pair of components that are three bits and six bits in length on a byte-addressable machine. The second component may be allocated at the fourth bit of the first byte or the first bit of the second byte.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: ADDITIONAL REPRESENTATION PRAGMAS

Question: What additional representation pragmas are provided by the implementation?

Reference: RM 13.1(13).1 - Explicit

Index Terms: *Representation Clauses*

Rationale:

For certain applications, implementation-defined pragmas may be necessary to support special requirements.

Example:

An implementation may define a representation pragma that allows the user to select from several representations of a subprogram, such as for non-reentrant subprograms.

TOPIC: EXPRESSIONS IN ENUMERATION REPRESENTATION CLAUSES

Question: What are the limitations on expressions that appear in enumeration representation clauses?

Reference: RM 13.3(1).1 - Implicit

Index Terms: *Representation Clauses*

Rationale:

The RM defines that the interpretation of the expressions that appear in representation clauses is implementation dependent. These interpretations are documented in Appendix F of the RM.

Example:

An implementation may have a convention that only allows unsigned integers to represent enumeration literals.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: LIMITATIONS ON RECORD ALIGNMENTS

Question: What are the restrictions on the allowable alignments for record representation clauses?

Reference: RM 13.4(4).1 - Explicit

Index Terms: *Representation Clauses*

Rationale:

The RM defines that an implementation is permitted to place restrictions on the allowable alignments for record representation clauses. These restrictions are documented in Appendix F of the RM.

Example:

Due to hardware restrictions an implementation may require that all record representation clauses have an alignment that places them on word boundaries.

TOPIC: ORDERING OF BITS IN RECORD REPRESENTATION CLAUSES

Question: What is the ordering of bits of storage units in component clauses within record representation clauses?

Reference: RM 13.4(5).1 - Explicit

Index Terms: *Representation Clauses*

Rationale:

The RM does not specify if the "first bit of a storage unit" refers to the least or most significant bit. Furthermore, it defines that the ordering of bits in storage units is machine dependent. The ordering of bits may affect the range of values used in component clauses within record representation clauses.

Example:

On some computer systems the most significant bit is the zeroth bit while on others the least significant bit is the zeroth bit. Alternatively, bits are sometimes numbered starting at one.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: NAMES OF IMPLEMENTATION-DEPENDENT RECORD COMPONENTS

Question: What are the conventions used for any implementation dependent components generated in records?

Reference: RM 13.4(8).1 - Explicit

Index Terms: *Records*

Rationale:

An implementation may generate names for implementation-dependent components that are added to a record. To select storage places for these implementation-dependent components in component clauses, it is necessary to know the naming conventions for these components.

Example:

An implementation may add an implementation-dependent component to hold the offset of another component within the record layout.

TOPIC: EXPRESSIONS IN ADDRESS CLAUSES

Question: What is the interpretation of expressions that appear in address clauses?

Reference: RM 13.5(3).1 - Explicit

Index Terms: - *Representation Clauses*

Rationale:

The RM defines that the interpretation of the expressions that appear in address clauses is implementation dependent. These interpretations are documented in Appendix F of the RM.

Example:

An implementation may have a convention that distinguishes the value of an expression as an address of a storage location or as an address of machine register.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: OVERLAYS

Question: What is the effect of using address clauses for overlaying objects, program units, or interrupts?

Reference: RM 13.5(8).1 - Explicit

Index Terms: *Representation Clauses, Erroneous Execution*

Rationale:

The RM defines that any program using address clauses to achieve the overlaying of objects, program units, or interrupts is erroneous. An implementation may provide some detection of this erroneous situation, either at compile-time or at run-time.

Example:

An error message at compile-time would be very useful in drawing attention to this hard-to-find error.

TOPIC: DEFINITION OF PACKAGE SYSTEM

Question: What is the definition of package SYSTEM?

Reference: RM 13.7(2).1 - Explicit

Index Terms: *Package System*

Rationale:

This is information that is basic for many applications.

Example:

The information is critical to developing very target dependent code, such as I/O device drivers or operating systems.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: MEANING OF THE ADDRESS REPRESENTATION ATTRIBUTE

Question: What is the meaning of the ADDRESS representation attribute?

Reference: RM 13.7.2(3).1 - Implicit

Index Terms: *Attributes*

Rationale:

For systems programming applications it is important to know the meaning for the ADDRESS attribute for various objects.

Example:

Consider the object of an access variable X. What is the value of X.ALL'ADDRESS when allocated objects may include additional storage for internal storage links? If the storage links are placed at the beginning of the allocated object, the ADDRESS attribute may point to the storage link as opposed to the actual value of the allocated object.

TOPIC: DEFINITION OF OTHER REPRESENTATION ATTRIBUTES

Question: What is the actual definition of the attributes: POSITION, FIRST_BIT, LAST_BIT and STORAGE_SIZE?

Reference: RM 13.7.2(5).1, 13.7.2(8-14).1 - Implicit

Index Terms: *Attributes*

Rationale:

The meaning for these attributes are unchanged from one implementation to another. But the actual values of these attributes may change from one implementation to another. Use of these attributes should be isolated to implementation or machine dependent program units.

Example:

For the same record type R and the same component C of R, the value of R.C'POSITION may be different from one implementation to another.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: ELABORATION CHECKS FOR INTERFACED SUBPROGRAMS

Question: Are elaboration checks performed for subprograms provided via the INTERFACE pragma?

Reference: RM 13.9(3).1, 3.9(5) - Implicit

Index Terms: *Elaboration, Runtime Checks*

Rationale:

AI-00180 states:

"If a subprogram is named in an INTERFACE pragma, no check need be made that the subprogram body has been elaborated before it is called."

Since no Ada body is provided for such a subprogram, it is implementation dependent as to whether or not the body need be elaborated. In some implementations it may be necessary to elaborate a non-Ada body, and therefore elaboration checks (possibly resulting in PROGRAM_ERROR) could be performed.

Example:

It may be possible to get a PROGRAM_ERROR when calling a subprogram that has a body provided with the INTERFACE pragma.

TOPIC: INTERFACE TO OTHER LANGUAGES

Question: Is the pragma INTERFACE supported and how is this interface supported?

Reference: RM 13.9(4).1 - Explicit

Index Terms: *Optimization, Foreign Languages*

Rationale:

This is critical feature for reusing existing software in other languages. While the RM only describes this feature with respect to the calling conventions to non-Ada software, an implementation will need to consider the implications of interfacing to these routines on other features of the Ada language.

Example:

An implementation will need to consider how an error that occurs when the non- Ada subprogram is executing is handled by the calling Ada program.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: RESTRICTIONS ON UNCHECKED TYPE CONVERSIONS

Question: What restrictions, if any, are imposed on unchecked type conversions by the implementation?

Reference: RM 13.10.2(2).1 - Explicit

Index Terms: *Type Conversions*

Rationale:

The RM permits an implementation to place restrictions on unchecked type conversions.

Example:

An implementation may not permit unchecked type conversions altogether, or it may limit it to types whose sizes are the same.

TOPIC: PRAGMA INTERFACE

Question: What programming languages (including various versions) are supported by the pragma INTERFACE?

Reference: RM Annex B(5).1 - Explicit

Index Terms: *Optimization (Time), Foreign Languages*

Rationale:

The RM allows an implementation to interface Ada programs to non-Ada subprograms through this pragma but it does not define which languages those may be. That is left up to the implementation.

Example:

An implementation which is available for a general-purpose computer that has FORTRAN, COBOL, and PASCAL may support the INTERFACE pragma for any one of these languages or none.

Catalogue of Ada Runtime Implementation Dependencies - Other Issues

TOPIC: PRAGMA INTERFACE

Question: What restrictions does the implementation place on the pragma INTERFACE?

Reference: RM Annex B(5).2 - Explicit

Index Terms: *Optimization (Time), Foreign Languages*

Rationale:

The RM does not require that all implementations provide this pragma. An implementation may place restrictions on the allowable forms and places of parameters and calls.

Example:

An implementation may require that all such calls to non-Ada programs through the use of the **pragma INTERFACE** be made only at the top level of the application, such as a library unit or the main subprogram.

TOPIC: DURATION REPRESENTATION

Question: What is the representation of the predefined type DURATION?

Reference: RM Annex C(19).1 - Implicit

Index Terms: *Time, Duration*

Rationale:

An implementation is free to choose the range and accuracy of values for the predefined type DURATION, within the restrictions that it must represent at least 86400 seconds and that DURATION'SMALL must not be greater than 20 milliseconds.

Example:

To comply with the language defined constraints on DURATION, it takes 24 bits of accuracy to represent values of DURATION. If the target machine hardware can support 32-bit fixed point arithmetic, an implementation may choose to support finer gradients of time.

Index

Abort

- Abort completion, p 51
- Mapping the tasking model onto an existing runtime, p 34
- Mechanism for abort completion, p 51
- Task termination after main program terminates, p 41
- The order of abortion, p 50

Accept

- Corresponding to an interrupt entry call, p 54
- Example, p 39
- Example, p 47
- Example, p 88
- Interrupt entry call, p 53
- Nested Rendezvous within an interrupt handler, p 56
- Open accept alternative, p 46
- Rendezvous optimizations, p 42
- With a terminate alternative, p 55

Access type

- Accessing unchecked deallocated objects, p 95
- Indivisible operations, p 52
- Input-Output of access type objects, p 103
- Storage reserved, p 26

Accessing

- Unchecked deallocated objects, p 95

Activation

- Effect of Priority, p 49

Activation order

- Of Tasks, p 38

ACVC

- Ada Compiler Validation Capability, p 2

Add immediate

- Representation of a literal, p 23

Address clauses

- Interpretation of expressions, p 130
- Overlays, p 131

Address representation

- Meaning of, p 132

Aggregate

- Evaluation of the component expression of a record aggregate, p 80
- Evaluation order of component associations, p 81
- Order of constraint checking, p 82
- Representation in the executable program, p 24

AI

- 00007 Discriminant Checks, p 74
- 00010 Expressions within Pragmas, p 120
- 00018 Order of constraint checking, p 82
- 00046 Closing of temporary files, p 106
- 00050 End of line and string, p 115
- 00180 Elaboration checks for INTERFACE subprograms, p 133
- 00181 NUMERIC_ERROR exception for nonstatic universal operations p 63
- 00189 Evaluation of component expressions, p 80
- 00200 Pragma Inline, p 16
- 00239 Representation of non-graphic characters, p 116
- 00278 Form Parameter, p 102
- 00288 Priority, Effect on activation, p 49
- 00320 Sharing External Files, p 113
- 00333 Assignment statement evaluation, p 84
- 00339 Allowable characters in comments, p 119
- 00357 Closing of sequential files, p 107
- 00365 Elaboration of generic instantiations, p 91
- 00387 Constraint Error in place of Numeric Error, p 64
- 00397 Exceptions during storage allocation, p 8
- 00399 Task termination, p 41
- AI-00239, p 118

Allocation

- Of a task object, p 40
- Representation of a literal, p 23
- Static allocation of pre-elaborated tasks, p 36
- Static storage allocation for objects, p 13

Index

- Static, elaboration prior to program execution, p 124
- Storage space allocation scheme for access collection, p 26
- Task activation, p 27
- Task storage allocation, p 37
- Appendix F
 - Expressions in address clauses, p 130
 - Expressions in enumeration representation clauses, p 27
 - Expressions in enumeration representation clauses, p 128
 - Limitations on record alignment, p 28
 - Limitations on record alignment, p 129
- Argument
 - Null FORM, p 104
- Arithmetic Operations
 - Causing overflow, p 20
- Array layout
 - Storage layout for array types, p 30
- Array type
 - Array length, p 125
 - Evaluation of a slice, p 79
 - Evaluation of an indexed component, p 78
 - Example p 100
 - Example, p 84
 - Impact when using pack, p 26
 - Index constraints, p 73
 - Index evaluation order, p 71
 - Order of evaluation, p 91
 - Order of the elaboration of component subtype indications, p 72
 - Passing array parameters, p 14
 - Selecting the representation, p 32
 - Storage layout, p 30
- Arrays
 - Evaluation order of component associations, p 81
 - Multidimensional, p 30
- ARTEWG
 - The Ada RunTime Environment Working Group, p
- Assignment statement
 - Constraint checking, p 125
 - Evaluation of, p 84
 - Implementation of an aggregate, p 24
 - Raising CONSTRAINT_ERROR, p 68
 - The order of evaluation for the operands, p 90
- Attributes
 - Definition of representation attributes, p 132
 - Image of non-graphic characters, p 118
 - Implementation-Defined, p 121
 - Meaning of the address representation attribute, p 132
- Base type
 - Using a wider range, p 65
 - Value outside the base type, p 58
- Bits
 - Allocation on a byte-addressable machine, p 32
 - Allocation on a byte-addressable machine, p 127
 - Ordering in record representation clauses, p 129
 - To represent type DURATION, p 135
- Buffering
 - File Input, p 114
 - File Output, p 114
- Case Statement
 - Criteria the compiler uses to make selection, p 10
- CHARACTER
 - Integer-to-Character conversions, p 25
 - Representation of non-graphic characters, p 116
- Character Set
 - Allowable characters in comments, p 119
- Code Motion
 - Error detection, efficiency, p 19
 - Optimizing code movement, p 100

Index

- Code sharing
 - For objects of the same task type, p 35
 - Shared code for generics, p 126
- Collection sizes
 - Formulas for calculating, p 26
- Comment
 - Allowable Characters, p 119
- Compilation units
 - Elaboration order, p 89
- Component
 - Elaboration order of component subtype, p 72
 - Evaluation of an indexed component, p 78
 - Evaluation order of component associations, p 81
 - Location of storage when not used, p 29
 - Minimizing the space between, p 127
 - Overlapping the storage boundary, p 28
 - associations of a record aggregate evaluation, p 80
- Component association
 - Order of constraint checking, p 82
- Component clauses
 - Within record representation clauses, p 129
- Component expressions
 - Evaluation of a record aggregate, p 80
- Components
 - Names of implementation-dependent record components, p 130
- Composite types
 - Passed as parameters, p 14
 - Size specifications, p 26
- Constraint checking
 - Assignment statement, p 125
 - Order of, p 82
- Constraints
 - Index, p 73
 - Mapping the tasking model onto an existing RTE, p 34
 - On the type DURATION, p 135
 - Order of constraint checking, p 82
 - Order of evaluation of index, p 73
 - Prohibiting boundary crossings, p 28
- CONSTRAINT_ERROR
 - Assignment statement constraint checking, p 125
 - Assignment statement evaluation, p 84
 - Code motion, p 100
 - Effect of incorrect order dependencies, p 68
 - Elaboration of block statements, p 12
 - Order of constraint checking, p 82
 - Size of a direct access file, p 31
 - Updating an out parameter, p 87
 - Value of scalar out parameters, p 122
- Context swap
 - Rendezvous optimizations, p 42
- Contiguous storage data
 - Aggregates represented in the executable code, p 24
- Conversions
 - Integer rounding, p 61
 - NUMERIC_ERROR exception for real conversions, p 60
 - Restrictions on unchecked type conversions, p 134
- Copy-Back
 - Order of parameters, p 87
 - Parameter passing mechanism, p 14
 - Value of scalar out parameters, p 122
- DATA_ERROR
 - Representation of non-graphic characters, p 116
 - Un-Interpretable elements, p 110
- Deallocated
 - Accessing unchecked objects, p 95
- Deallocation
 - Of storage for tasks that have terminated, p 37

Index

- Default expressions
 - Evaluation of, p 69
- Delay alternatives
 - Algorithm used to select from delay, p 47
- Delay resolution
 - Execution accuracy, p 44
- Delays
 - Delay alternatives, p 47
 - Delay resolution, p 44
 - Evaluation of delay expression or entry family index, p 46
 - Scheduling event caused by delay statement, p 43
 - The type Time, p 45
- Direct access file
 - The maximum size, p 31
- Discriminant
 - Assignment statement constraint checking, p 125
 - Assignment statement evaluation, p 84
 - Erroneous assignment, p 94
 - Evaluation order, p 76
- Distributed Ada
 - Package STANDARD, p 123
- Distributed computer systems
 - Code sharing, p 35
- Duration
 - Representation, p 135
- DURATION'SMALL
 - Consistency with the delay resolution, p 44
 - DURATION representation, p 135
- Dynamic tracking
 - Exception handler, p 18
- Elaboration
 - Activation order of tasks, p 38
 - Component subtype elaboration order, p 72
 - Discriminant Checks, p 74
 - Elaboration checks, p 6
 - Elaboration checks, p 77
- Elaboration of Block Statements, p 12
- Generic instantiations, p 91
- Order of compilation units, p 89
- Pre-elaboration of tasks, p 36
- Prior to program execution, p 124
- checks for INTERFACE subprograms, p 133
- Embedded system
 - Main program termination, p 99
- Embedded systems
 - Exception representation, p 96
 - Scheduling order of tasks, p 49
- End of Line
 - Effect of, p 115
- End of String
 - Effect of, p 115
- Entry
 - Nested Rendezvous within an interrupt handler, p 56
 - Of an accept statement, p 54
 - With a terminate alternative, p 55
- Entry calls
 - Delay resolution, p 44
 - Direct execution of, p 53
 - Priority of interrupts, p 54
- Entry family index
 - Evaluation of, p 46
- Entry names
 - Elaboration order of generic instantiations, p 91
- Enumeration
 - Representation clauses, p 27
 - Representation clauses, p 128
- ENUMERATION_IO
 - Representation of non-graphic characters, p 116
- Erroneous execution
 - Accessing unchecked deallocated objects, p 95
 - Assignment statement evaluation, p 84

Index

- Effect of, p 92
 - Evaluation of scalar variables, p 93
 - Overlays, p 131
 - PROGRAM_ERROR, p 94
 - Shared variables, p 52
- Error detection, efficiency
- Interaction with the optimizer, p 19
- Exception handler
- Dynamic tracking and static mapping, p 18
 - Generated inline, p 20
- Exceptions
- Assignment statement constraint checking, p 125
 - Associated overhead when raising an exception, p 17
 - Caused by use of the optimizer, p 19
 - Code motion, p 100
 - Code optimization, p 65
 - Discriminant Checks, p 74
 - During storage allocation, p 8
 - Exception handler overhead, p 18
 - File IO not supported, p 105
 - Handled through hardware, p 20
 - Handler overhead, p 98
 - Implementation-Defined, p 97
 - Implementation-defined, p 101
 - Main program termination, p 99
 - NUMERIC_ERROR for nonstatic universal operations, p 63
 - NUMERIC_ERROR for real conversions, p 60
 - NUMERIC_ERROR for real operations, p 59
 - NUMERIC_ERROR raised for an operand in an expression, p 58
 - Non-Ada, p 99
 - PROGRAM_ERROR, p 90
 - PROGRAM_ERROR, p 94
 - Pragma SUPPRESS, p 20
 - Pragma SUPPRESS, p 101
 - Representation of non-graphic characters, p 116
 - Representation, p 96
 - STORAGE_ERROR during task creation/activation, p 40
 - Sharing external files, p 104
 - Suppressed, p 20
 - Un-Interpretable elements, p 110
 - Updating an out parameter, p 87
- Execution order
- Task activation, p 39
- Expressions
- Component associations in an array aggregate evaluation, p 81
 - Constraint checks before all the choices or expressions are evaluated, p 82
 - Default, p 69
 - Evaluation of operands, p 83
 - Explicit generic actual parameters or constituents of variable names or entry names, p 91
 - In a indexed component, p 78
 - In a slice, p 79
 - In address clauses, p 130
 - In enumeration representation clauses, p 27
 - In enumeration representation clauses, p 128
 - In representation clauses, p 126
 - Of a range constraint, p 70
 - Of a record aggregate, p 80
 - Of an open delay alternative, p 46
 - Order of evaluation as parameters, p 86
- External file
- Control shared access, p 102
 - Default options, p 104
 - Deletion of temporary files, p 21
 - Deletion of, p 108
 - File IO not supported, p 105
 - Name associated with a temporary file, p 109
 - Sequential, p 107
 - Sharing, p 104
 - Status, p 103
 - Temporary, p 106
 - Waiting for page terminators, p 112
- External file status
- Disposition, p 103
- Float
- Exception handler overhead, p 98
 - Pragma INLINE, p 16
 - Storage layout for array types, p 30

Index

- Floating point
 - Assignment statement evaluation, p 84
 - Error detection, p 19
 - NUMERIC_ERROR exception for nonstatic universal operations, p 63
 - Overflow, p 64
 - Pragma SUPPRESS, p 101
 - Raising NUMERIC_ERROR exception for an operand in an expression, p 58
 - Representation, p 67
 - Short_Float and Long_Float representations, p 67
- Foreign languages
 - Interface to other languages, p 133
 - Non-Ada exceptions, p 99
 - Pragma INTERFACE, p 134
 - Pragma INTERFACE, p 135
- Form Parameter
 - System dependent characteristics, p 102
- Format
 - Ada runtime implementation dependency issues are shown, p 4
- Garbage collection schemes
 - Reclamation of objects created by allocators which are no longer accessible, p 7
- Generic
 - Elaboration of instantiations, p 91
 - Shared code, p 126
- Get
 - Representation of non-graphic characters, p 116
- Get_Line
 - End of line and string, p 115
 - Waiting for page terminators, p 112
- Guard condition
 - Evaluation, p 88
- Image
 - Of a non-graphic character, p 118
 - Size of code for shared generics, p 126
- Implementation Dependent
 - Accuracy of static real expressions, p 62
 - Comparisons of real operands, p 59
 - Expressions in address clauses, p 130
 - Expressions in enumeration representation clauses, p 27
 - Expressions in enumeration representation clauses, p 128
 - Names of record components, p 130
 - Storage reserved for a collection size, p 26
 - Storage reserved for task activation, p 27
- Implementation-defined
 - Additional representation pragmas, p 128
 - Attributes, p 121
 - Exceptions, p 97
 - Exceptions, p 101
 - Non-model numbers, p 59
 - Non-model numbers, p 62
- In out
 - Main program parameters and results, p 124
 - Order of parameter copy-back, p 87
- Index
 - Evaluation order, p 71
- Index constraints
 - Order of evaluation, p 73
- Indexed component
 - Evaluation of, p 78
- Initialization
 - Evaluation of default expressions, p 69
 - ask termination when the main program terminates, p 41
- Input-Output
 - Access type objects, p 103
 - Buffering, p 114
 - Closing of sequential files, p 107
 - Closing of temporary files, p 106
 - Deletion of shared external files, p 108
 - End of line and string, p 115
 - External file status, p 103
 - File IO not supported, p 105
 - Form parameter, p 102
 - Image of non-graphic characters, p 118
 - Implementation defined exceptions, p 101

Index

- Name of an external file associated with a temporary file, p 109
- Null form argument string, p 104
- Representation of non-graphic characters, p 116
- Sharing External Files, p 113
- Sharing external files, p 104
- Size of direct access file, p 31
- Temporary file status, p 21
- Terminators, p 111
- Un-Interpretable elements, p 110
- Waiting for page terminators, p 112
- Instantiation
 - Elaboration of generics, p 91
 - Shared code for generics, p 126
- Integer Type
 - Predefined, p 66
 - Representation, p 66
 - Rounding for conversions, p 61
 - Shared code for generics, p 126
- INTEGER'LAST
 - Compared to the direct access file size, p 31
- Integer_IO
 - Put procedure to do integer-to-character conversion, p 25
- Interface
 - Pragma INTERFACE, p 30
 - Pragma INTERFACE, p 134
 - Pragma INTERFACE, p 135
 - To an assembly language program, p 21
 - To other languages, p 133
- Interleaving algorithms
 - In a monoprocessor, p 33
- Interrupt entry calls
 - Direct execution, p 53
 - Priority, p 54
- Interrupt handler
 - With rendezvous nested within, p 56
- Interrupt handlers
 - The suspension of, p 54
- Interrupt vector
 - Nested Rendezvous within an interrupt handler, p 56
- Length clause
 - Control afforded on an access type, p 26
 - Impact on packing algorithm, p 26
 - control afforded on task activation, p 27
- Library unit
 - Elaboration order of compilation units, p 89
 - Pragma INTERFACE, p 135
 - Unreferenced elements of a package in the executable code, p 25
- Limitations
 - On record alignments, p 28
 - On record alignments, p 129
- Literal
 - Represented in the executable program, p 23
- LONG_FLOAT
 - Representation, p 67
- LONG_INTEGER
 - Representation, p 66
- Machine code inserts
 - Package MACHINE_CODE, p 21
- Machine level operations
 - Using the pragma INLINE, p 21
- MACHINE_OVERFLOWS
 - To raise NUMERIC_ERROR, p 59
 - To raise NUMERIC_ERROR, p 60
- Main program
 - Default priority, p 48
 - Direct execution of interrupt entry calls, p 53
 - External file status after termination of, p 103
 - Initiation, p 123
 - Parameter restrictions, p 124
 - Result restrictions, p 124
 - Task termination after termination of, p 41

Index

- Temporary file status after termination of, p 21
- Termination, p 99
- Mapping
 - From the exception name to an underlying representation, p 96
 - Static, p 18
 - The tasking model onto an existing RTE, p 34
- Multiple processors
 - Task scheduling, p 48
- Non-Ada
 - Exceptions, p 99
 - Interface to other languages, p 133
 - NON_ADA_ERROR in package SYSTEM, p 97
 - Pragma INTERFACE, p 134
 - Pragma INTERFACE, p 135
 - Runtime overlays, p 29
- Non-closed file
 - External file status, p 103
- Non-Graphic characters
 - Image of, p 118
 - Representation, p 116
- Non-shared memory
 - Code sharing, p 35
- Nonsharing
 - Of files using form parameters, p 102
- Nonstatic Universal operations
 - NUMERIC_ERROR, p 63
- Null character
 - Representation of non-graphic characters, p 116
- Null FORM
 - Argument string, p 104
- Null statement
 - Code generated for the null statement, p 9
 - Runtime penalty imposed, p 9
- Numerics
 - Accuracy of static real expressions, p 62
 - Code optimization, p 65
 - Comparisons of real operands, p 59
 - Evaluation of default expressions, p 69
 - Evaluation of operands in an expression, p 83
 - Float representation, p 67
 - Index constraints, p 73
 - Index evaluation order, p 71
 - Integer representation, p 66
 - NUMERIC_ERROR exception for nonstatic universal operations, p 63
 - NUMERIC_ERROR exception for real conversions, p 60
 - NUMERIC_ERROR exception for real operations, p 59
 - NUMERIC_ERROR, p 57
 - NUMERIC_ERROR, p 64
 - Raising NUMERIC_ERROR exception for an operand in an expression, p 58
 - Range constraint evaluation, p 70
 - Rounding for integer conversions, p 61
 - SHORT_FLOAT and LONG_FLOAT representation, p 67
 - SHORT_INTEGER and LONG_INTEGER representation, p 66
 - NUMERIC_ERROR
 - Code motion, p 100
 - Code optimization, p 65
 - Exception for real conversions, p 60
 - Exception for real operations, p 59
 - Exceptions for nonstatic universal operations, p 63
 - For an operand in an expression p 58
 - Non-Ada exceptions, p 99
 - On an intermediate operation, p 57
 - Supported in hardware, p 64
 - Suppressing checks, p 20
- Obsolete
 - 14.2.1(9).2 Closing Sequential Files p 107
 - 14.3.6(13).1 End of line and string, p 115
- Open accept alternative
 - Evaluation of, p 46
- Open alternatives
 - Selective wait alternatives, p 47

Index

- Open delay alternative
 - Evaluation of, p 46
- Operands
 - Comparison of real operands, p 59
 - Evaluation of operands in an expression, p 83
 - Exception handler overhead, p 98
 - The order of evaluation, p 90
- Optimization
 - Interface to other languages, p 133
 - Pragma SUPPRESS, p 20
- Optimization (Space)
 - Code sharing for objects of the same task type, p 35
 - Library unit (Unreferenced elements of), p 25
 - Overlapping storage boundaries in record representation clauses, p 28
 - Packing algorithm for pragma PACK, p 127
 - Representation of aggregates, p 24
 - Shared code for generics, p 126
 - Size specification for composite types, p 26
 - Temporary file status, p 21
- Optimization (Time and Space)
 - Case statement implementation, p 10
 - Exception representation, p 96
 - Null statement, p 9
 - Parameter passing conventions, p 15
 - Parameter passing mechanism, p 14
 - Pragma INLINE, p 16
 - Pragma OPTIMIZE, p 32
 - Pragma SHARED, p 52
 - Representation of a literal, p 23
 - Task storage allocation, p 37
- Optimization (Time and Storage)
 - Pragma PACK, p 32
- Optimization (Time)
 - Code motion, p 100
 - Code optimization, p 65
 - Elaboration checks, p 6
 - Elaboration of Block Statements, p 12
 - Evaluation of delay expression or entry family index, p 46
 - Exception handler overhead, p 18
 - Exception handler overhead, p 98
 - Exception raise overhead, p 17
 - Input-Output of access type objects, p 103
 - Machine code inserts, p 21
 - Pragma INLINE, p 22
 - Pragma INTERFACE, p 134
 - Pragma INTERFACE, p 135
 - Pragma SUPPRESS, p 20
 - Pragma SUPPRESS, p 22
 - Pragma SUPPRESS, p 101
 - Pre-elaboration of tasks, p 36
 - Rendezvous optimizations, p 42
 - Scheduling algorithm for tasking, p 33
 - Storage Reclamation, p 7
 - Storage allocation for objects, p 13
 - The cause of an exception, p 19
 - The type Time, p 45
- Optimizer
 - Performing a subtype check, p 125
 - The cause of an exception, p 19
- Or
 - Evaluation causing an exception, p 100
- Order Dependence
 - Activation order of tasks, p 38
 - Assignment statement constraint checking, p 125
 - Assignment statement evaluation, p 84
 - Component subtype elaboration order, p 72
 - Discriminant Checks, p 74
 - Discriminant evaluation order, p 76
 - Effect of being incorrect, p 68
 - Elaboration checks, p 77
 - Elaboration of generic instantiations, p 91
 - Elaboration order of compilation units, p 89
 - Evaluation of a slice, p 79
 - Evaluation of an indexed component, p 78
 - Evaluation of default expressions, p 69
 - Evaluation of delay expression or entry family index, p 46
 - Evaluation of operands in an expression, p 83
 - Evaluation of the component expression of a record aggregate, p 80
 - Evaluation order of component associations, p 81
 - Guard condition evaluation, p 88

Index

- Index constraints, p 73
- Index evaluation order, p 71
- Order of abortion, p 50
- Order of constraint checking, p 82
- Order of evaluation of parameter associations, p 86
- Order of parameter copy-back, p 87
- PROGRAM_ERROR, p 90
- Range constraint evaluation, p 70
- Scheduling order of tasks, p 49
- Selective wait alternatives, p 47
- Task activation (execution order), p 39
- Out
 - Order of parameter copy-back, p 87
 - Value of scalar parameter, p 122
- Overflow
 - Error detection, efficiency, p 19
 - NUMERIC_ERROR exception for nonstatic universal operations, p 63
 - NUMERIC_ERROR exception for real conversions, p 60
 - NUMERIC_ERROR exception for real operations, p 59
 - NUMERIC_ERROR, p 64
 - Raising NUMERIC_ERROR, p 20
- OVERFLOW_CHECK
 - NUMERIC_ERROR, p 64
- Overhead
 - Associated with interrupt control may be substantial, p 52
 - Associated with suppressing a check, p 20
 - Associated with the elaboration of a block statement, p 12
 - Associated with type Time, p 45
 - Concerning execution of the raise statement, p 17
 - Elaboration prior to program execution, p 124
 - Exception handler, p 98
 - Exception handlers, p 18
 - Rendezvous optimization, p 42
 - Scheduling event caused by delay statement, p 43
 - Storage allocation for objects, p 13
- Overlays
 - Effect of using address clauses, p 131
 - Runtime overlays, p 29
- Package body
 - Unreferenced elements in the executable program, p 25
- Package MACHINE_CODE
 - Machine code inserts, p 21
- Package STANDARD
 - Compilation of distributed Ada, p 123
- Package SYSTEM
 - Definition, p 131
 - Exception handling, p 97
- Packing Algorithm
 - Impact when used with the length clause, p 26
 - Pragma PACK, p 127
- Parameter
 - (out) value of scalar, p 122
 - FORM, p 102
 - Main program parameters and results, p 124
 - Order of copy-back, p 87
 - Order of evaluation, p 86
 - Passing conventions, p 15
 - Passing mechanism, p 14
- Parameter passing mechanism
 - Copy-in vs. copy-back, p 14
- Performance
 - Evaluation of delay expression or entry family index, p 46
 - Exception handler overhead, p 18
 - Exceptions during storage allocation, p 8
 - Implementation defined exceptions, p 101
 - Input-Output of access type objects, p 103
 - Of an executable program, p 6
 - Parameter passing mechanism, p 14
 - Pragma SUPPRESS, p 20
 - Pragma shared, p 52
 - Rendezvous between tasks without priorities, p 50
 - Shared code for generics, p 126
 - Storage Reclamation, p 7

Index

- Portability
 - Acceptance of representation clauses, p 127
 - Expressions in representation clauses, p 126
- PRAGMA
 - Expressions within, p 120
- Pragma INLINE
 - Machine code subprogram, p 21
 - Subprograms expanded inline, p 16
 - Subprograms not expanded inline, p 22
- Pragma INTERFACE
 - Interface to other languages, p 133
 - Languages supported, p 134
 - Restrictions, p 135
 - Storage layout for array types, p 30
- Pragma OPTIMIZE
 - Case statement implementation, p 10
 - The effect of using and not using, p 32
- Pragma PACK
 - Packing algorithm, p 127
 - The effect on an Ada program, p 32
- Pragma SHARED
 - Implemented for supported types, p 52
 - Shared variables, p 52
- Pragma SUPPRESS
 - Circumstances when ignored p 20
 - It's effect, p 22
 - Overhead associated with suppressing a check, p 20
 - The effect, p 101
- Pragmas
 - Additional representation pragmas, p 128
 - For specification of program overlays, p 29
- Pre-elaboration of tasks
 - Tasks in a library level package statically allocated, p 36
- Preemptive priority
 - Interleaving algorithm, p 33
- Preemptive scheduling
 - Priority, Effect on activation, p 49
 - Task scheduling on multiprocessors, p 48
- Priorities
 - Activation order of tasks, p 38
 - Effect on task activation, p 49
 - Evaluation of delay expression or entry family index, p 46
 - Rendezvous between tasks without priorities, p 50
 - Rendezvous optimizations, p 42
 - Scheduling event caused by delay statement, p 43
 - Task Priorities, p 48
 - Task activation (execution order), p 39
 - Task scheduling on multiprocessors, p 48
 - The order of abortion, p 50
- Priority of interrupt entry calls
 - Suspension of interrupt handlers, p 54
- Program initiation
 - Elaboration prior to program execution, p 124
 - Main program initiation, p 123
 - Main program parameters and results, p 124
- Program termination
 - Main program termination, p 99
- PROGRAM_ERROR
 - Due to incorrect order dependencies, p 90
 - Effect of incorrect order dependencies, p 68
 - Guard condition evaluation, p 88
 - Raised because of erroneous program, p 94
 - Value of scalar out parameters, p 122
- Put
 - Representation of non-graphic characters, p 116
- Raise statement
 - Associated overhead, p 17
- Range constraint
 - Order of evaluation of simple expressions, p 70

Index

- Real**
 - Accuracy of static real expressions, p 62
- Real conversions**
 - NUMERIC_ERROR exception, p 60
- Real Operands**
 - Comparisons, p 59
- Real-time**
 - Applications that execute the delay statement, p 44
 - Applications that use tasking, p 42
 - Applications that use type Time, p 45
 - Applications using accept statement, p 53
 - Elaboration of block statements, p 12
- Record**
 - Bit ordering in representation clause, p 129
 - Component which has no component clause, p 29
 - Permitting a component to overlap a storage boundary, p 28
 - The storage layout, p 29
 - representation clauses, p 28
 - representation clauses, p 129
- Record aggregate**
 - Evaluation of the component expressions of, p 80
- Record representation clause**
 - Limitations on record alignment, p 28
 - Limitations on record alignment, p 129
 - Ordering of bits, p 129
 - Overlapping storage boundaries, p 28
 - Record component with no component clause, p 29
- Records**
 - Evaluation of the component expression of a record aggregate, p 80
 - Names of implementation-dependent components, p 130
- Rendezvous**
 - (Nested) within an interrupt handler, p 56
 - Available optimizations, p 42
 - Between tasks without priorities, p 50
- Representation**
 - Integer, p 66
- Representation attributes**
 - Definition of, p 132
- Representation clause**
 - Size specification for composite types, p 26
 - Storage reserved for a collection size, p 26
- Representation clauses**
 - Acceptance of, p 127
 - Additional representation pragmas, p 128
 - Bit ordering, p 129
 - Direct execution of interrupt entry calls, p 53
 - Expressions in address clauses, p 130
 - Interpretation of expressions, p 126
 - Limitations on expressions, p 27
 - Limitations on expressions, p 128
 - Limitations on record alignment, p 28
 - Limitations on record alignment, p 129
 - Nested Rendezvous within an interrupt handler, p 56
 - Overlapping storage boundaries, p 28
 - Overlays, p 131
 - Packing algorithm for pragma PACK, p 127
 - Priority of interrupt entry calls, p 54
 - Storage layout for record types, p 29
 - Storage reserved for task activation, p 27
- Representation of a literal**
 - Representation in the executable program, p 23
- Representation of aggregates**
 - Representation in the executable program, p 24
- Response-critical**
 - Applications using task priorities, p 48
- Restrictions**
 - Of DURATION representation, p 135
 - On terminate alternatives, p 55
 - On the allowable alignments for record representation clauses, p 28
 - On the allowable alignments for record representation clauses, p 129

Index

- On the parameters and results of a main program, p 124
- On the pragma INTERFACE, p 135
- On unchecked type conversions, p 134
- Reusability
 - Buffering file input and output, p 114
 - Closing of sequential files, p 107
 - Closing of temporary files, p 106
 - Deletion of shared external files, p 108
 - Dependence upon maximum size for a direct access file, p 31
 - End of line and string, p 115
 - File IO not supported, p 105
 - Input-Output of access type objects, p 103
 - Name of an external file associated with a temporary file, p 109
 - Null form argument string, p 104
 - Representation of non-graphic characters, p 116
 - Sharing External Files, p 113
 - Sharing external files, p 104
 - Terminators, p 111
 - Un-Interpretable elements, p 110
 - Waiting for page terminators, p 112
- Reusable
 - Form parameter, p 102
- RM
 - 01.06(07).01 Effect of erroneous execution, p 92
 - 01.06(09).01 Effect of incorrect order dependencies, p 68
 - 02.08(07).01 Expressions within Pragmas, p 120
 - 03.02.01(15).01 Evaluation of default expressions, p 69
 - 03.02.01(17).01 Evaluation of scalar variables, p 93
 - 03.02.01(18).01 Evaluation of scalar variables, p 93
 - 03.05(05).1 Range constraint evaluation, p 70
 - 03.05.04(10).01 Numeric_error, p 57
 - 03.06(10).01 Index evaluation order, p 71
 - 03.06(10).02 component subtype elaboration order, p 72
 - 03.06.01(11).11 Index constraints, p 73
 - 03.07.02(05).01 Discriminant Checks, p 74
 - 03.07.02(13).01 Discriminant evaluation order, p 76
 - 03.09(04).01 Elaboration checks, p 6
 - 03.09(05).01 Elaboration checks, p 77
 - 04.01.01(04).01 Evaluation of an indexed component, p 78
 - 04.01.02(04).01 Evaluation of a slice, p 79
 - 04.01.04(04).01 Implementation defined attributes, p 121
 - 04.02(01).01 Representation of a literal, Chapter 5 p 23
 - 04.02(01).01 Representation of a literal, p 119
 - 04.03(01).01 Aggregates (representation of), p 24
 - 04.03.01(03).01 Evaluation of component expressions, p 80
 - 04.03.02(10).1 Evaluation order of component associations, p 81
 - 04.03.02(11).01 Order of constraint checking, p 82
 - 04.05(05).01 Evaluation of operands in an expression, p 83
 - 04.05(07).01 Raising numeric_error exceptions for an operand in an expression, p 58
 - 04.05.07(07).01 Numeric_error exception for real operations, p 59
 - 04.05.07(10).01 Comparisons of real operands, p 59
 - 04.06(07).01 Numeric_error exception for real conversions, p 60
 - 04.06(07).02 Rounding for integer conversions, p 61
 - 04.08(07).01 Storage Reclamation, p 7
 - 04.08(13).01 Exceptions during storage allocation, p 8
 - 04.09(12).01 Accuracy of static real expressions, p 62
 - 04.10(05).01 NUMERIC_ERROR exception for nonstatic universal operations p 63
 - 05.01(05).01, Null Statement, p 9
 - 05.02(03).01 Assignment statement evaluation, p 84
 - 05.02(04).01 Assignment statement evaluation, p 84
 - 05.04(01).01 Case statement implementation, p 10
 - 05.06(04).01 Elaboration of block statements, p 12

Index

- | | |
|---|--|
| <p>06.01(10).01 Storage allocation for objects, p 13</p> <p>06.02(05).01 Value of scalar out parameters, p 122</p> <p>06.02(07).01 Parameter passing mechanism, p 14</p> <p>06.02(07).02 Parameter passing conventions, p 15</p> <p>06.03.02(04).01 Pragma Inline, p 16</p> <p>06.04(06).01 Order of evaluation of parameter associations, p 86</p> <p>06.04(06).02 Order of parameter copy-back, p 87</p> <p>08.06(01).01 Package Standard, p 123</p> <p>09(02).01 Scheduling algorithm for tasking, p 33</p> <p>09(02).2 Mapping the tasking model onto an existing runtime, environment, p 34</p> <p>09.01(06).01 Code sharing for objects of the same task type, p 35</p> <p>09.02(02).01 Pre-elaboration of tasks, p 36</p> <p>09.02(02).02 Task storage allocation, p 37</p> <p>09.03(01).01 Activation order of tasks, p 38</p> <p>09.03(02).01 Task activation (execution order), p 39</p> <p>09.03(03).01 Storage_error during task creation/activation, p 40</p> <p>09.04(13).01 Task termination, p 41</p> <p>09.05(14).01 Rendezvous optimizations, p 42</p> <p>09.06(01).01 Scheduling event caused by delay statement, p 43</p> <p>09.06(04).01 Delay resolution, p 44</p> <p>09.06(05).01 The type time, p 45</p> <p>09.07.01(05).01 Guard condition evaluation, p 88</p> <p>09.07.01(05).02 Evaluation of delay expression or entry family index, p 46</p> <p>09.07.01(06).01 Selective wait alternatives, p 47</p> <p>09.07.01(08).01 Delay alternatives, p 47</p> <p>09.08(01).01 Task priorities, p 48</p> <p>09.08(04).01 Task scheduling on multiple processors, p 48</p> <p>09.08(04).02 Priority, Effect on activation, p 49</p> <p>09.08(05).01 Scheduling order of tasks, p 49</p> <p>09.08(05).02 Rendezvous between tasks without priorities, p 50</p> <p>09.10(04).01 Order of abortion, p 50</p> | <p>09.10(06).01 Mechanism for abort completion, p 51</p> <p>09.10(08).01 Abort Completion, p 51</p> <p>09.11(08).01 Shared variables, p 52</p> <p>09.11(11).1 Pragma Shared, p 52</p> <p>10.01(08).01 Main program initiation, p 123</p> <p>10.01(08).02 Main program parameters and results, p 124</p> <p>10.04(01).01 Library unit (unreferenced elements of), p 25</p> <p>10.05(02).01 Elaboration order of compilation units, p 89</p> <p>10.05(02).02 Elaboration prior to program execution, p 124</p> <p>11.01(01).01 Exception representation, p 96</p> <p>11.01(04).01 Implementation-defined exceptions, p 97</p> <p>11.01(06).01 Numeric_error, p 64</p> <p>11.01(07).01 Program_error, p 94</p> <p>11.01(07).02 Program_error, p 90</p> <p>11.02(03).01 Exception handler overhead, p 98</p> <p>11.02(06).01 Non-Ada exceptions, p 99</p> <p>11.03(03).01 Exception raise overhead, p 17</p> <p>11.04(01).01 Exception handler overhead, p 18</p> <p>11.04.01(05).01 Main program termination, p 99</p> <p>11.04.01(20).01 Main program termination, p 99</p> <p>11.06(04).01 Assignment statement constraint checking, p 125</p> <p>11.06(05).01 Error detection, efficiency, p 19</p> <p>11.06(06).01 Code optimization, p 65</p> <p>11.06(11).01 Code motion, p 100</p> <p>11.07(18).01 Pragma Suppress, p 101</p> <p>11.07(18).02 Pragma Suppress, p 20</p> <p>11.07(20).01 Pragma Suppress, p 20</p> <p>12.03(05).01 Shared code for generics, p 126</p> <p>12.03(17).01 Elaboration of generic instantiations, p 91</p> <p>13.01(10).01 Expressions in representation clauses, p 126</p> <p>13.01(10).02 Acceptance of representation clauses, p 127</p> |
|---|--|

Index

- 13.01(12).01 Packing algorithm for the pragma Pack, p 127
- 13.01(13).01 Additional representation pragmas, p 128
- 13.02(05).01 Size specification for composite types, p 26
- 13.02(08).01 Storage reserved for a collection size, p 26
- 13.02(10).01 Storage reserved for a task activation, p 27
- 13.03(01).01 Expressions in enumeration representation clauses, p 27
- 13.03(01).01 Expressions in enumeration representation clauses, p 128
- 13.04(04).01 Limitations on record alignments, p 28
- 13.04(04).01 Limitations on record alignments, p 129
- 13.04(05).01 Ordering of bits in record representation clauses, p 129
- 13.04(05).02 Overlapping storage boundaries in record representation clauses, p 28
- 13.04(06).01 Storage layout for record types, p 29
- 13.04(08).01 Names of implementation-dependent record components, p 130
- 13.05(03).01 Expressions in address clauses, p 130
- 13.05(08).01 Overlays, p 131
- 13.05(10).01 Runtime overlays, p 29
- 13.05.01(02).01 Direct execution of interrupt entry calls, p 53
- 13.05.01(02).02 Priority of interrupt entry calls, p 54
- 13.05.01(03).01 Restrictions on terminate alternative, p 55
- 13.05.01(05).01 Nested rendezvous within an interrupt handler, p 56
- 13.07(02).01 Definition of Package System, p 131
- 13.07.02(03).01 Meaning of the address representation attribute, p 132
- 13.07.02(05).01 Definition of other representation attributes, p 132
- 13.07.02(08).01 Definition of other representation attributes, p 132
- 13.07.02(09).01 Definition of other representation attributes, p 132
- 13.07.02(10).01 Definition of other representation attributes, p 132
- 13.07.02(11).01 Definition of other representation attributes, p 132
- 13.07.02(12).01 Definition of other representation attributes, p 132
- 13.07.02(13).01 Definition of other representation attributes, p 132
- 13.07.02(14).01 Definition of other representation attributes, p 132
- 13.08(04).01 Machine code inserts, p 21
- 13.08(05).01 Machine code inserts, p 21
- 13.08(06).01 Machine code inserts, p 21
- 13.09(04).01 Interface to other languages, p 133
- 13.09(06).01 Storage layout for array types, p 30
- 13.10.01(06).01 Accessing unchecked deallocation objects, p 95
- 13.10.02(02).01 Restrictions on unchecked type conversion, p 134
- 13.9(3).1 Elaboration checks for INTERFACE subprograms, p 133
- 14.01(01).01 Form Parameter, p 102
- 14.01(07).01 External file status, p 103
- 14.01(07).02 Input-output of access type objects, p 103
- 14.01(11).01 Implementation-defined exceptions, p 101
- 14.01(13).01 Sharing external files, p 104
- 14.02.01(03).01 Size of a direct access file, p 31
- 14.02.01(03).02 Temporary file status, p 21
- 14.02.01(03).03 Null form argument string, p 104
- 14.02.01(03).04 File IO not supported, p 105
- 14.02.01(13).01 Deletion of shared external files, p 108
- 14.02.01(21).01 Name of an external file associated with a temporary file, p 109
- 14.02.01(9).01 Closing of temporary files, p 106
- 14.02.01(9).02 Closing of sequential files, p 107
- 14.02.02(04).01 Un-interpretable elements, p 110
- 14.02.04(04).01 Un-interpretable elements, p 110
- 14.03(07).01 Terminators, p 111

Index

- 14.03.04(08).01 Waiting for page terminators, p 112
- 14.03.05(03).01 Sharing External Files, p 113
- 14.03.05(13).01 Waiting for page terminators, p 112
- 14.03.06(01).01 Buffering file input and output, p 114
- 14.03.06(07).01 Buffering file input and output, p 114
- 14.03.06(13).01 End of line and string, p 115
- 14.03.09(06).01 Representation of non-graphic characters, p 116
- 14.03.09(09).01 Representation of non-graphic characters, p 116
- Ada Reference Manual, p 2
- Annex A(18).1 Image of non-graphic characters, p 118
- Annex B(04).01 Pragma Inline, p 22
- Annex B(05).01 Pragma Interface, p 134
- Annex B(05).02 Pragma Interface, p 135
- Annex B(08).01 Pragma Optimize, p 32
- Annex B(09).01 Pragma Pack, p 32
- Annex B(14).01 Pragma Suppress, p 22
- Annex C(06).01 Integer representation, p 66
- Annex C(07).01 Short integer & long integer representations, p 66
- Annex C(09).01 Float representation, p 67
- Annex C(10).1 Short float and long float representations, p 67
- Annex C(19).1 Duration representation, p 135

- ROMable Code
 - Static allocation, p 36

- Rounding
 - For integer conversions, p 61

- Run-till-blocked
 - Interleaving algorithm, p 33

- Runtime Checking
 - Error detection, efficiency, p 19
 - Order of constraint checking, p 82
 - PROGRAM_ERROR, p 94
 - Pragma SUPPRESS, p 20
 - Pragma SUPPRESS, p 20
 - Pragma SUPPRESS, p 22

- Pragma SUPPRESS, p 101

- Runtime Checks
 - Assignment statement constraint checking, p 125
 - Elaboration checks, p 6
 - Elaboration checks, p 77
 - NUMERIC_ERROR exception for nonstatic universal operations, p 63
 - NUMERIC_ERROR exception for real conversions, p 60
 - NUMERIC_ERROR exception for real operations, p 59
 - NUMERIC_ERROR, p 57
 - NUMERIC_ERROR, p 64
 - Raising NUMERIC_ERROR exception for an operand in an expression, p 58

- Runtime Environment
 - Implementation defined attributes, p 121
 - Mapping a tasking model onto it, p 34
 - Rendezvous optimizations, p 42
 - Task priorities, p 48

- Runtime overlays
 - Pragmas for specification, p 29

- Scalar variables
 - Evaluation of, p 93

- Scheduling
 - Algorithm for tasking, p 33
 - Order of tasks, p 49
 - Priority, Effect on activation, p 49
 - Scheduling event caused by delay statement, p 43
 - Task priorities, p 48
 - Task scheduling on multiprocessors, p 48

- Scheduling algorithm
 - For tasking, p 33

- Segmented memory
 - Code sharing, p 35

- Select
 - Guard condition evaluation, p 88
 - With a terminate alternative, p 55

Index

- Select statement
 - Evaluation of delay expression or entry family index, p 46
- Selective wait
 - Algorithm to select from the open alternatives, p 47
 - Delay alternatives, p 47
 - Guard condition evaluation, p 88
- Shared code
 - Generics, p 126
- Shared External file
 - Deletion of, p 108
- Shared Variables
 - Used within tasks, p 52
- Sharing
 - External files, p 104
 - Files using form parameter, p 102
- SHORT_FLOAT
 - Representation, p 67
- SHORT_INTEGER
 - Representation, p 66
- SIGAda
 - Special Interest Group, p
- Size Specification
 - For Composite types, p 26
- Skip_Line
 - End of line and string, p 115
 - Waiting for page terminators, p 112
- Slice
 - Effect of incorrect order dependencies, p 68
 - Evaluation of, p 79
 - Program error, p 90
 - Time slice interleaving algorithm, p 33
- Static
 - Accuracy of static real expressions, p 62
- Static allocation
 - Elaboration prior to program execution, p 124
 - Pre-elaboration of tasks, p 36
 - Representation of a literal, p 23
 - Storage allocation for objects, p 13
- Static mapping
 - Exception handler, p 18
- Storage
 - Boundaries overlapping for representation clauses, p 28
 - Layout for array types, p 30
 - Reserved for a collection size, p 26
 - Reserved for task activation, p 27
 - Runtime overlays, p 29
 - STORAGE_ERROR during task creation/activation, p 40
 - Size specification for composite types, p 26
 - The layout for record types, p 29
- Storage (File size)
 - Size of direct access file, p 31
- Storage allocation
 - For tasks, p 37
 - Interaction with constraint_error, p 8
 - Local objects allocated statically, p 13
 - STORAGE_ERROR during task creation/activation, p 40
- Storage Management
 - Exceptions during storage allocation, p 8
 - Storage Reclamation, p 7
- Storage Reclamation
 - Time the storage reclamation is performed, p 7
- STORAGE_ERROR
 - During task creation/activation, p 40
- Store immediate
 - Representation of a literal, p 23
- Subcomponent
 - Evaluation of scalar variables, p 93

Index

- Subprogram
 - Expanded inline using pragma `INLINE`, p 22
 - Expanded inline without using Pragma `INLINE`, p 22
 - Library unit (Unreferenced elements of), p 25
 - Machine level operations, p 21
 - Non-Ada, p 133
 - Non-ada, p 99
 - Order of evaluation of parameter associations, p 86
 - Order of parameter copy-back, p 87
 - Parameters, p 14
 - Parameters, p 15
 - Parameters, p 17
 - Parameters, p 124
 - Recursive, p 37
 - Using Pragma `INLINE`, p 16
 - Value of scalar out parameters, p 122
- Subprogram Invocation
 - Pragma `INLINE`, p 16
- Subprogram Parameters
 - Exception raise overhead, p 17
 - Parameter passing conventions, p 15
 - Parameter passing mechanism, p 14
- Subtype
 - Assignment statement constraint checking, p 125
 - Component elaboration order, p 72
 - Range constraint evaluation, p 70
- Suppress
 - Associated overhead, p 20
- `SYSTEM.MAX_INT`
 - `NUMERIC_ERROR` exception for nonstatic universal operations, p 63
- `SYSTEM.MIN_INT`
 - `NUMERIC_ERROR` exception for nonstatic universal operations, p 63
- `SYSTEM.TICK`
 - Accuracy, p 45
 - Consistency with the delay resolution, p 44
- Task activation
 - Execution order, p 39
 - `STORAGE_ERROR`, p 40
 - Storage reserved, p 27
- Task Creation
 - `STORAGE_ERROR`, p 40
- Task management scheme
 - For the activation of a task, p 27
- Task priorities
 - Implementation characteristics, p 48
- Task scheduling
 - On multiple processors, p 48
 - Priority, Effect on activation, p 49
- Task Type
 - Activation order of tasks, p 38
 - Code sharing, p 35
- Tasking
 - Abort completion, p 51
 - Activation order, p 38
 - Code sharing for objects of the same task type, p 35
 - Delay alternatives, p 47
 - Delay resolution, p 44
 - Direct execution of interrupt entry calls, p 53
 - Evaluation of delay expression or entry family index, p 46
 - Guard condition evaluation, p 88
 - Mapping the tasking model onto an existing RTE, p 34
 - Mechanism for abort completion, p 51
 - Nested Rendezvous within an interrupt handler, p 56
 - Pragma `SHARED`, p 52
 - Pre-elaboration of tasks, p 36
 - Priority of interrupt entry calls, p 54
 - Priority, Effect on activation, p 49
 - Rendezvous between tasks without priorities, p 50
 - Rendezvous optimizations, p 42
 - Restrictions on terminate alternative, p 55
 - `STORAGE_ERROR` during task creation/activation, p 40
 - Scheduling algorithm, p 33

Index

- Scheduling event caused by delay statement, p 43
- Scheduling order of tasks, p 49
- Selective wait alternatives, p 47
- Shared variables, p 52
- Sharing external files, p 104
- Storage allocation, p 37
- Storage reserved for task activation, p 27
- Task activation (execution order), p 39
- Task priorities, p 48
- Task scheduling on multiprocessors, p 48
- Task termination when the main program terminates, p 41
- The order of abortion, p 50
- Tasking model
 - Mapping onto an existing RTE, p 34
- Temporary File Status
 - After completion of main program, p 21
- Terminate
 - A task when the main program terminates, p 41
 - Restrictions, p 55
- Termination
 - Main program, p 99
- Terminators
 - The effects of, p 111
 - Waiting for page terminators, p 112
- Text_IO
 - Included in program using no input or output, p 25
- Time slice
 - Interleaving algorithm, p 33
- Time-critical
 - Applications using task priorities, p 48
- Transportability
 - Buffering file input and output, p 114
 - Closing of sequential files, p 107
 - Closing of temporary files, p 106
 - Deletion of shared external files, p 108
 - Dependence upon maximum size for a direct access file, p 31
 - End of line and string, p 115
 - External file status, p 103
 - File IO not supported, p 105
 - Form parameter, p 102
 - Implementation defined exceptions, p 101
 - Input-Output of access type objects, p 103
 - Name of an external file associated with a temporary file, p 109
 - Null form argument string, p 104
 - Representation of non-graphic characters, p 116
 - Sharing External Files, p 113
 - Sharing external files, p 104
 - Terminators, p 111
 - Un-Interpretable elements, p 110
 - Waiting for page terminators, p 112
- Type conversion
 - Rounding for integer conversions, p 61
 - Unchecked, p 134
- Type Time
 - Associated overhead, p 45
- Un-Interpretable elements
 - Effect of, p 110
- Unchecked
 - Deallocated objects, p 95
- Unchecked type conversions
 - Restrictions, p 134
- Undefined values
 - Evaluation of scalar variables, p 93
 - Value of scalar out parameters, p 122
- Unmaskable interrupt
 - Associated with fixed point overflow, p 20
- USE_ERROR
 - File IO not supported, p 105
 - Input-Output of access type objects, p 103
 - Null form argument string, p 104
- Variables
 - Shared, p 52
- With clause
 - Partial ordering, p 89

Index

Word boundary
Alignment of a record, p 28
Alignment of a record, p 129

Index

|